HAW HAMBURG

MASTER THESIS
Hans Hartmann

Abstraktion in Event-Streaming-Verarbeitungssystemen

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Hans Hartmann

Abstraktion in Event-Streaming-Verarbeitungssystemen

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang *Master of Science Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Olaf Zukunft
Zweitgutachter: Prof. Dr. Ulrike Steffens

Eingereicht am: 27.03.2023

Hans Hartmann

Thema der Arbeit

Abstraktion in Event-Streaming-Verarbeitungssystemen

Stichworte

Event-Steaming, Apache Spark, Kafka-Streams, Apache Storm, Apache Flink, KsqlDB, Apache Beam, Abstraktion, Data Abstraktion, Datenverarbeitung

Kurzzusammenfassung

Eine große Anzahl der heutigen Anwendungsfälle für Datenverarbeitung arbeitet mit kontinuierlichen und unendlichen Daten. Um diese Daten zu verarbeiten, existieren viele Event-Streaming-Verarbeitungssysteme mit unterschiedlichen Eigenschaften und Abstraktionen. Das Ziel dieser Masterarbeit ist es, die Abstraktionen von Apache Spark, Apache Storm, Kafka-Streams, KsqlDB, Apache Flink und Apache Beam zu untersuchen. Dafür wird ein Experiment in den Event-Streaming-Verarbeitungssystemen durchgeführt und ausgewertet. Für das Experiment wird ein Job in den genannten Systemen implementiert, welcher dieselben Aufgaben erfüllt und gleichzeitig Latenzzeiten misst. Das Experiment testet die Event-Streaming-Verarbeitungssysteme auf ihre Latenz sowie Komplexität. Zunächst wird das Grundkonzept von der Arbeit mit kontinuierlichen Daten sowie die verschiedenen Event-Streaming-Verarbeitungssysteme vorgestellt. Anschließend wird erläutert, wie das Experiment umgesetzt worden ist. Die Ergebnisse zeigen, dass Abstraktionen die Komplexität dieser Systeme reduzieren können. Durch Abstraktionen ist es möglich, dass der Fokus der Entwickler auf der Funktionalität der Anwendung liegt und nicht auf den darunterliegenden Details wie Skalierbarkeit oder Fehlertoleranz. Abstraktionen können sich auch negativ auf die Leistung eines Systems auswirken, in Apache Beam brauchen die abstrahierten Systeme länger für das Lesen, Schreiben und Verarbeiten der Daten als die nativen Systeme. ...

Hans Hartmann

Title of Thesis

Abstraction in Event-Streaming-Processing-Systems

Keywords

Event-Steaming, Apache Spark, Kafka-Streams, Apache Storm, Apache Flink, KsqlDB, Apache Beam, Abstraction, Data abstraction, Processing Data

Abstract

A large number of today's use cases for data processing works with continuous and infinite data. Many event streaming processing systems exist to process this data, with different characteristics and abstractions. The goal of this master thesis is to evaluate the abstractions of Apache Spark, Apache Storm, Kafka Streams, KsqlDB, Apache Flink and Apache Beam. For this purpose, an experiment is executed and evaluated in these event streaming processing systems. For the experiment, a job is implemented in the above-mentioned systems, which performs the same tasks and measures latency at the same time. The experiment tests the event streaming processing systems for their latency as well as their complexity. First, the basic concept of working with continuous data and the different event streaming processing systems are introduced. Then, it is explained how the experiment has been implemented and the results are presented. The results show that abstractions can reduce complexity. Through abstractions it is possible that the focus of the developers is on the functionality of the application rather than on the underlying details such as scalability or fault tolerance. Abstractions can also have a negative impact on the performance of a system, in Apache Beam, abstracted systems take longer to read, write, and process data than the native systems. ...

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Aufgrund des steigenden Volumens von kontinuierlichen Daten und dem Wissen, welches aus diesen Daten abgeleitet werden kann, erhalten skalierbare und effiziente Event-Streaming-Verarbeitungssysteme immer mehr an Bedeutung.

Die Verarbeitung von kontinuierlichen Daten hat sich in den letzten Jahren enorm entwickelt und weist ein großes Potenzial auf. Firmen können anhand der erhobenen Daten von z.B. Anwendungen, Logs oder Sensoren in nahezu Echtzeit Informationen durch die Verarbeitung dieser Daten ableiten, welches für das Geschäft und die Kunden sehr nützlich sein kann.

Derzeit gibt es viele Systeme für die Aufnahme, Verarbeitung, Speicherung und Verwaltung von kontinuierlichen Daten, wodurch die Auswahl der richtigen Kombination von Tools und Plattformen für den Aufbau von Stream-Verarbeitungs-Anwendungen nicht leicht ist. [29] Es existieren viele Event-Streaming-Verarbeitungssysteme mit stark unterschiedlichen Fähigkeiten und Leistungsmerkmalen [46] wie Apache Spark oder Apache Storm, die dabei helfen große und kontinuierliche Daten zu verarbeiten.

In der Entwicklung werden Abstraktionen dafür eingesetzt, dass Programmierer den Code wiederverwenden können und keinen 'Low-Level-Code' schreiben müssen. Abstraktionen helfen dabei, die Komplexität zu reduzieren, indem die Schnittstelle von den Implementierungsdetails getrennt wird. Programmierfunktionen, wie zum Beispiel die Funktion pow() in Python, sind ein gutes Beispiel für Abstraktionen. Wenn die Funktion pow() in Python verwendet wird, um die Potenz einer Zahl zu berechnen, wird die Funktion im Hintergrund ausgeführt, ohne dass der Anwender den genauen Algorithmus kennen muss.

In Event-Streaming-Verarbeitungssystemen werden verschiedene Abstraktionen eingesetzt.

Zum Beispiel durch die Verwendung von Datenabstraktionen, High-Level-Programmiermodellen und Schnittstellen. Diese Modelle und Schnittstellen bieten Entwicklern eine vereinfachte Sicht auf die Infrastruktur der Event-Streaming-Verarbeitungssysteme, sodass diese sich auf die Implementierung der Anwendungslogik konzentrieren können. Entwickler müssen sich dadurch nicht mit Details auf einer niedrigeren Ebene wie dem Datenrouting, dem Lastausgleich und der Fehlertoleranz befassen. Beispiele für solche High-Level-Programmiermodelle sind z.B. Apache Kafka Streams, Apache Flink oder Apache Beam.

Abstraktion ist ein Schlüsselkonzept in Event-Streaming-Verarbeitungssystemen, welche sich erheblich auf die Implementierung und den Betrieb auswirken kann. Dies kann dazu beitragen, die Entwicklung und den Einsatz von Stream-Verarbeitungs-Anwendungen zu vereinfachen und ihre Skalierbarkeit, Zuverlässigkeit und Leistung zu verbessern. Abstraktionen können sich jedoch auch negativ auf die Leistung des Systems auswirken. In Event-Streaming-Verarbeitungssystemen werden Abstraktionen an vielen verschiedenen Stellen eingesetzt.

1.1 Motivation

Abstraktionen in Event-Streaming-Verarbeitungssystemen haben einen hohen Stellenwert. Eine sehr wichtige Abstraktion in diesen Systemen ist die Datenabstraktion. In Anwendungen, die kontinuierliche Daten verarbeiten, müssen die Daten zunächst in eine Datenstruktur abstrahiert werden. Unterschiedliche Event-Streaming-Verarbeitungssysteme benutzen verschiedene Implementierungen der Datenabstraktion. Durch die Abstraktion der Daten ist es möglich, diese zu verarbeiten, wodurch verschiedene Funktionen wie ein Filter oder Transformationen auf den Daten angewendet werden können. Verschiedene Abstraktionen haben unterschiedliche Vor- sowie Nachteile. Die Fehlertoleranz, Skalierbarkeit und Performance sind wichtige Kriterien für die Datenabstraktion. Es gibt mehrere Event-Streaming-Verarbeitungssysteme, die für unterschiedliche Anwendungsfälle geeignet und optimiert sind. Die bekanntesten sind unter anderem Apache Spark und Apache Flink.

Die Übertragung einer Implementierung von einem Stream-Verarbeitungssystem auf ein anderes z.B. aus Leistungsgründen erfordert die Anpassung bestehender Anwendungen an neue Schnittstellen. [31] Apache Beam ermöglicht, durch eine Abstraktionsschicht denselben Code auf unterschiedlichen Ausführungsumgebungen auszuführen. Die-

se Abstraktionsschicht abstrahiert die verschiedenen Funktionen der Event-Streaming-Verarbeitungssysteme und führt diese in einer Umgebung zusammen. Diese Systeme können durch die Benutzung von Abstraktionen unterschiedlich lange für dieselben Aufgaben benötigen.

Daher werden die verschiedenen Abstraktionen in Event-Streaming-Verarbeitungssystemen evaluiert.

1.2 Ziel

Das Ziel dieser Masterarbeit ist es, verschiedene Abstraktionen in Event-Streaming-Verarbeitungssystemen zu evaluieren. Um Abstraktionen in Event-Streaming Systemen näher zu betrachten, werden verschiedene Fragestellungen behandelt. Es soll die Frage beantwortet werden, wie komplex und aufwändig es ist, einen Stream-Verarbeitungsjob in den verschiedenen Event-Streaming-Verarbeitungssystemen zu implementieren. Erleichtert eine hohe Abstraktionsschicht die Implementierung der Streaming-Jobs in den Verarbeitungssystemen? Des Weiteren soll evaluiert werden, wie die Events in den verschiedenen Stream-Verarbeitungssystemen abstrahiert werden? In z.B. Spark existieren mehrere Möglichkeiten, die Daten zu abstrahieren. Hier soll beantwortet werden, welche Abstraktion für welchen Anwendungsfall am besten geeignet ist und wie die Datenstruktur intern aufgebaut ist. Anschließend soll geprüft werden, wie hoch die Kosten für die Abstraktionen in den Event-Streaming-Verarbeitungssystemen sind. Auf den Daten werden Berechnungen durchgeführt, die unterschiedlich implementiert sind. Die Fragen, wie schnell die Daten verarbeitet und Antworten erhalten werden, sollen beantwortet werden. Abschließend wird Apache Beam analysiert. Apache Beam abstrahiert mehrere Ausführungsumgebungen, wodurch es möglich wird, denselben Code auf mehrere Ausführungsumgebungen auszuführen. Es soll begutachtet werden, wie Apache Beam die verschiedenen Funktionen der Ausführungsumgebungen abstrahiert und zu einem System zusammenführt. Zusätzlich wird analysiert, wie die Latenzzeiten von einem nativen Event-Streaming-Verarbeitungssystem wie z.B. Apache Flink im Gegensatz zu der Ausführungsengine des Flink Runners von Apache Beam sind.

1.3 Aufbau der Arbeit

Die folgenden Abschnitte geben einen kurzen Überblick über diese Arbeit und ihren Aufbau:

Zunächst werden im zweiten Kapitel die **Grundlagen von Event-Streaming-Verarbeitungssystemen** erläutert. Dies beinhaltet die Datenabstraktion, Zeit, Windows, Watermark und Trigger.

Das dritte Kapitel **Event-Streaming-Verarbeitungssysteme** beschäftigt sich mit den verschiedenen Event-Streaming-Verarbeitungssystemen. Hier werden die Abstraktionen und Schnittstellen sowie die Geschichte der Systeme erläutert.

Das vierte Kapitel befasst sich mit **Apache Beam**. In diesem Kapitel wird Apache Beam vorgestellt und erläutert, wie Apache Beam funktioniert.

Das **Experiment** wird im fünften Kapitel vorgestellt. Die Implementation und Umsetzung wird in diesem Abschnitt behandelt und erklärt.

Anschließend folgt eine **Diskussion**, in welcher die Ergebnisse des Experimentes vorgestellt und ausgewertet werden. Am Ende wird ein **Fazit** gezogen und ein **Ausblick** gegeben.

2 Grundlagen

Die Verarbeitung großer Datenmengen im Batch-Verfahren ist oft nicht geeignet für Anwendungen, in denen konitinuierliche Daten schnell verarbeitet werden müssen. Aus diesem Grund hat die Stream-Verarbeitung von kontinuierliche Daten eine große Aufmerksamkeit erlangt.[39]

In diesem Kapitel werden die Grundlagen des Streaming Konzeptes dargestellt.

2.1 Abstraktionen in Event-Streaming-Verarbeitungssystemen

Abstraktion wird als eine der grundlegenden Prinzipien des Software-Engineering bezeichnet, um die Komplexität zu beherrschen, werden nach[41] die unnötigen Details vor dem Anwender verborgen.

In Event-Streaming-Verarbeitungssystemen gibt es viele verschiedene Abstraktionen, welche die Trennung zwischen Konzept und Umsetzung ermöglichen. Strukturen werden dabei über ihre Bedeutung definiert, während die detaillierten Informationen über die Funktionsweise verborgen bleiben. Abstraktion zielt darauf ab, die Details der Implementierung nicht zu berücksichtigen und daraus ein allgemeines Schema zur Lösung des Problems abzuleiten.

Viele Ebenen von Abstraktionen werden in Event-Streaming-Verarbeitungssystemen eingesetzt, wie zum Beispiel die Datenabstraktion, welche eine klare Trennung zwischen den abstrakten Eigenschaften eines Datentyps und den konkreten Details seiner Implementierung ermöglicht. Zudem gibt es in einzelnen Event-Streaming-Verarbeitungssystemen verschiedene Schnittstellen, mit welchen Daten verarbeitet werden können. In Apache Spark gibt es z.B. die Spark Streaming API und die Spark Structured Streaming API.[51] Die beiden Schnittstellen sind vom Spark Core unterschiedliche abstrahiert und bieten

dem Anwender dadurch verschiedenen Datenabstraktionen, Funktionen und Möglichkeiten, um die Daten zu lesen, schreiben und zu verarbeiten.

Des Weiteren ist es möglich, Pipelines in gewissen Stream-Verarbeitungssystemen mit unterschiedlichen Programmiersprachen zu implementieren. So z.B in Apache Flink, es ist möglich, eine Flink Pipeline mit Java oder auch mit Python zu entwickeln. Pyflink wird mithilfe von Py4J in Java übersetzt, sodass die Flink Engine die Pipeline ausführen kann. Py4J kann als eine Brücke zwischen Java und Python bezeichnet werden. Py4J läuft in der Java Virtual Machine und ermöglicht, dass Python auf Java Klassen und Objekte zugreifen kann, als ob sich diese im Python-Interpreter befinden würden.[4]

Datenabstraktion

Die Datenabstraktion schafft eine klare Trennung zwischen den abstrakten Eigenschaften eines Datentyps und den konkreten Details seiner Implementierung. Die abstrakten Eigenschaften sind diejenigen, welche für den Anwender sichtbar sind, wie zum Beispiel die Schnittstelle zum Datentyp. Die konkrete Implementierung wird vom Anwender verborgen. So ist es möglich, einen abstrakten Datentyp zu definieren, welcher Werte mit Schlüsseln verknüpft. Dieser Datentyp kann mit verschiedenen Datenstrukturen implementiert werden, wie zum Beispiel mit einer Hashtabelle oder einer einfachen linearen Liste, welche Schlüssel-Wert Paare enthalten. Dies ermöglicht, dass die Darstellung von Daten lokal geändert werden können, ohne dass Programme, die diese Daten verwenden, davon betroffen sind. Datenabstraktionen sind besonders wichtig, weil dadurch komplizierte Dinge verborgen werden, die sich in der Zukunft ändern können. Außerdem vereinfachen sie die Struktur der Programme, da sie eine Schnittstelle auf höherer Ebene darstellen.[44]

Datenbanksysteme und Event Stream-Verarbeitungssysteme bestehen aus komplexen Datenstrukturen. Um die Interaktion mit den Daten zu vereinfachen, werden interne relevante Details vor dem Nutzer versteckt. Dieser Prozess wird Datenabstraktion genannt. Dies kann die Entwicklung von Anwendungen, welche Stream-Verarbeitungssystemen benutzen, vereinfachen, da sich die Entwickler auf die Anwendungslogik konzentrieren können, die mit den Daten selbst arbeitet, anstatt sich um die spezifischen Details zu kümmern wie die Fehlertoleranz, Skalierbarkeit oder auch die Funktionen wie Daten verarbeitet oder gespeichert werden.

2.2 Time

Bei der Verarbeitung von kontinuierlichen Daten hat die Zeit einen wichtigen Einfluss auf die Verarbeitung und die Korrektheit der Ergebnisse. Bei Stream-Verarbeitungssystemen wird eine unendliche Menge von Daten in ein System eingelesen, verarbeitet und wieder ausgegeben.

In diesen Systemen gibt es zwei wichtige Zeiteinheiten:

- Event-Time
- Processing-Time

Event-Time

Die Event-Time ist der Zeitpunkt, an dem das Ereignis tatsächlich eingetreten ist. Dieser Zeitstempel wird in den Meta-Daten eines Ereignisses festgehalten und vom System gesetzt, welches das Ereignis generiert hat.[2]

Processing-Time

Die Processing-Time (Verarbeitungszeit) ist der Zeitpunkt, an dem ein Ereignis an einem bestimmten Punkt, während der Verarbeitung, in der Pipeline beobachtet wird. Es sollte beachtet werden, dass die Uhrensynchronisation in einem verteilten System nicht einheitlich sein kann. Dies liegt z.B. an verschiedenen Zeitzonen.[2]

Event vs Processing Time

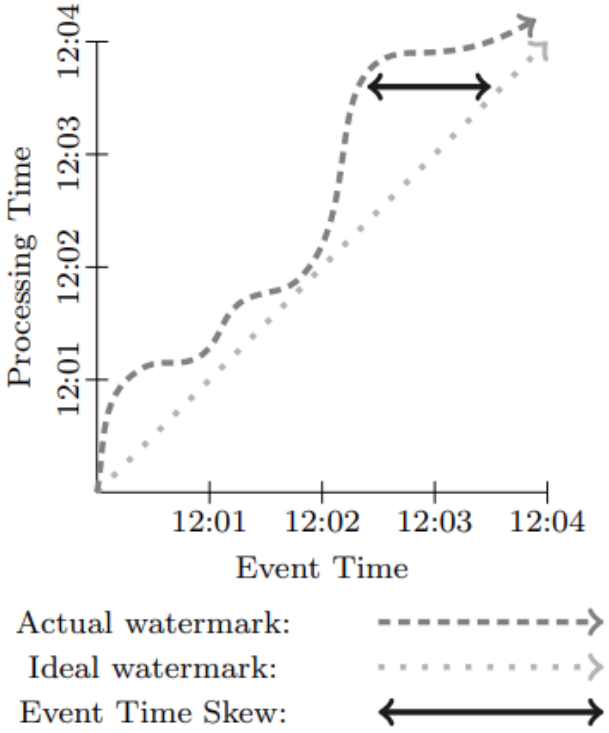

Abbildung 2.1: Event-Time vs Processing-Time[2]

In verteilten Systemen besteht eine Diskrepanz zwischen der Event- und Processing-Time.[2] Dieses Missverhältnis kann beliebige Verzögerungen beim Erhalt einer Antwort auf der Grundlage der Semantik der Ereigniszeit bedeuten.

Die Event-Time für ein bestimmtes Ereignis ändert sich praktisch nie, aber die Verarbeitungszeit ändert sich kontinuierlich für jedes Ereignis, wenn es die Pipeline durchläuft. Dies ist ein wichtiger Unterschied, wenn es um die Analyse von Ereignissen im Kontext des Zeitpunkts ihres Auftretens geht. Während der Verarbeitung führen die Funktionen der verwendeten Systeme (Verarbeitung, Kommunikation, Serialisierung etc.) zu einem sich dynamisch verändernden Verzerrungsgrad der beiden Zeiteinheiten. In einer idealen Welt würde das Event-Streaming-Verarbeitungssystem das Ereignis empfangen, wenn es eintritt. In Wirklichkeit kann die Abweichung zwischen dem Auftreten eines Ereignis-

ses und der Verarbeitung dieses Ereignisses durch das System jedoch stark variieren.[2] Siehe: 2.1

2.3 Window

Bei kontinuierlichen Daten müssen bei einigen Funktionen diese Daten in 'Windows' unterteilt werden, damit diese verarbeitet werden können. Dies ist z.B. bei Aggregationen der Fall.[2] Durch Windows werden die unendlichen und kontinuierlichen Daten in Fenster von endlichen Sammlungen entsprechend den Zeitstempeln der einzelnen Elemente gruppiert. Durch die Einteilung der unendlichen und kontinuierlichen Daten in Windows wird der Datensatz endlich und kann verarbeitet werden. Anschließend kann eine Aggregation auf der Teilmenge der kontinuierlichen Daten erfolgen.

Eine Window-Funktion teilt dem System mit, wie Elemente in einem Fenster zugeordnet werden. Es gibt verschiedene Varianten, um die kontinuierlichen Daten in sogenannte Windows zu unterteilen:

- Global Window
- Fixed Window
- Session Window
- Sliding Window

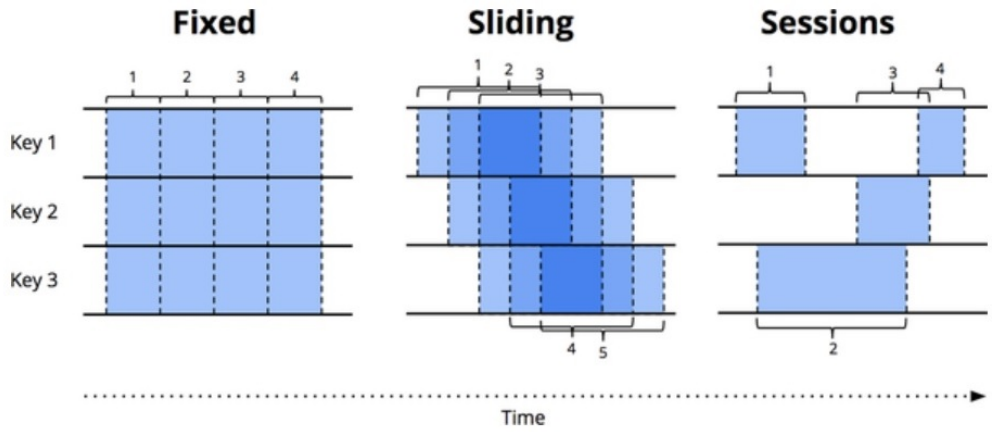

Abbildung 2.2: Windows[3]

Global Window

Ein Global-Window ist ein Fenster, welches sich auf den gesamten Zeitraum bezieht. Siehe: 2.3 Bei konitinuierlichen Daten gibt es kein Ende der Daten und auch keinen gesamten Zeitraum. Daher muss, wenn ein Global-Window in einem Stream-Verarbeitungssystem eingesetzt wird, ein Trigger implementiert werden Ansonsten würde das System keine Ergebnisse zurückgeben. Trigger werden im nächsten Abschnitt definiert. Siehe: 2.4

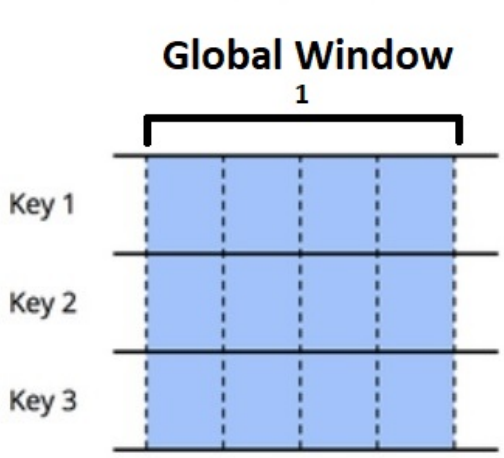

Abbildung 2.3: Global Window

Fixed Window

Beim Fixed-Window wird die Zeit in gleichmäßige, sich nicht überschneidende Abschnitte unterteilt. Jedes Ereignis gehört zu genau einem Fenster und jedes Fenster wird separat verarbeitet.[2] Siehe: 2.2

Session Window

Ein Session-Window beginnt mit dem Auftreten des ersten Ereignisses eines Schlüssels. Beim Session-Window wird eine Zeitspanne als Parameter übergeben, die festlegt, wie

lange keine neuen Daten eintreffen dürfen, bis das Window geschlossen und verarbeitet wird. Wenn innerhalb der angegebenen Zeitspanne ein weiteres Ereignis in das Fenster fällt, wird das Fenster um das neue Ereignis erweitert. Wenn innerhalb der Zeitüberschreitung keine Ereignisse zu dem jeweiligen Schlüssel auftreten, wird das Fenster nach Ablauf der Zeitüberschreitung geschlossen.

Falls weiterhin Ereignisse zu dem Schlüssel eintreffen, wird das Window so lange erweitert, bis die maximale Dauer erreicht ist. Siehe: 2.2. Die maximale Dauer und die Zeitspanne wird der session Window Funktion als Parameter übergeben.[2]

Sliding Window

In einem Sliding-Window werden Tupel innerhalb des Windows gruppiert, welche in einem bestimmten Zeitintervall in das Verarbeitungssystem gelangt sind. Der Sliding-Window Funktion wird ein Gleitintervall und eine Zeiteinheit übergeben. Die Zeiteinheit gibt an, wie lange auf Ereignisse gewartet werden soll, bis das Fenster geschlossen wird und der Gleitintervall definiert, wie oft das Window ausgewertet werden soll.[2] Ein zeitbasiertes Sliding-Window mit einer Länge von zehn Sekunden und einem Gleitintervall von fünf Sekunden enthält Tupel, die innerhalb von zehn Sekunden eintreffen. Die Menge der Tupel innerhalb des Fensters werden alle fünf Sekunden ausgewertet. Sliding-Windows können überschneidende Daten enthalten. Ein Ereignis kann beim Sliding-Window zu mehreren Windows gehören. Siehe: 2.2

2.4 Triggering

Durch Windows hat man die Möglichkeit, einen Wert aus einem Fenster zu erhalten, jedoch muss noch entschieden werden, wann die Berechnung ausgeführt werden soll.

Trigger steuern, wann die Elemente für einen bestimmten Schlüssel und ein bestimmtes Fenster ausgegeben werden. Wenn Elemente eintreffen, werden diese durch die Window Funktion in ein oder mehrere Fenster unterteilt und an den zugehörigen Trigger übergeben, um zu bestimmen, ob der Inhalt des Fensters berechnet werden soll.[2]

Ein Trigger ist eine Abstraktion von Richtlinien, die angeben, wie oft eine Streaming-Abfrage ausgeführt und möglicherweise neue Daten ausgegeben werden sollen.

Beim Definieren eines Triggers sollte die Frage beantwortet werden, wann die Berechnung ausgelöst werden soll.[2]

Es gibt verschiedene Varianten von Triggern:

- Daten basierte Trigger wie das Auslösen einer Berechnung nach mindestens n Ereignissen.
- Event time Trigger. Diese Auslöser basieren auf der Event-Time, welche in den Meta-Daten festgehalten wird.
- Processing time Trigger. Diese Auslöser werden anhand der Verarbeitungszeit, der Zeitpunkt, an dem das Ereignis in einer bestimmten Phase der Pipeline verarbeitet wird, ausgelöst.

[8]

Es ist in manchen Systemen möglich, mehrere Trigger miteinander zu kombinieren.[8]

Trigger bieten zwei zusätzliche Möglichkeiten im Vergleich zur einfachen Ausgabe am Ende eines Fensters:

- Mit Triggern können frühzeitig Ergebnisse ausgeben werden, bevor alle Daten in einem bestimmten Fenster eingetroffen sind. Die Ausgabe erfolgt zum Beispiel nach Ablauf einer bestimmten Zeit oder nach dem Eintreffen einer bestimmten Anzahl von Elementen.
- Trigger ermöglichen die Verarbeitung von spät eintreffenden Daten. Nachdem das Wasserzeichen(Watermark) für die Ereigniszeit das Ende eines Fensters überschritten worden ist, können Trigger dafür sorgen, dass eine neue Berechnung ausgelöst wird.

Diese Funktionen ermöglichen es, den Datenfluss zu kontrollieren und je nach Anwendungsfall zwischen verschiedenen Faktoren abzuwägen:

- Vollständigkeit: Wie wichtig ist es, dass alle Daten vollständig sind, bevor die Ergebnisse berechnet werden?
- Latenzzeit: Wie lange soll auf die Daten gewartet werden?

- Kosten: Wie viel Rechenleistung/Ressourcen sollen verwendet werden, um die Latenzzeit zu verringern?

[8]

Ein System, das zeitkritische Aktualisierungen erfordert, könnte beispielsweise einen zeitbasierten Auslöser verwenden, der alle N Sekunden ein Fenster ausgibt. Bei diesem Anwendungsfall ist die Schnelligkeit wichtiger als die Vollständigkeit der Daten. Ein System, das mehr Wert auf die Vollständigkeit der Daten als auf das exakte Timing der Ergebnisse legt, könnte sich für den Standard-Trigger entscheiden, der am Ende eines Fensters ausgelöst wird.

2.5 Watermark

In jedem Datenverarbeitungssystem gibt es eine gewisse Verzögerung zwischen dem Zeitpunkt, zu dem ein Ereignis im Stream-Verarbeitungssystem eintritt(Event-Time) und dem Zeitpunkt, zu dem das tatsächliche Datenelement in irgendeiner Phase in der Pipeline verarbeitet wird (Processing-Time). Es gibt keine Garantien darüber, dass die Ereignisse in der Pipeline in der gleichen Reihenfolge erscheinen, in der diese erzeugt worden sind. Watermark bezeichnet die Handhabung, wie verspätete Ereignisse verarbeitet werden und probiert, das allgemeine Problem zu lösen, zu welchem Zeitpunkt es sicher ist, ein Event-Time-Window als geschlossen zu bezeichnen. Dies bedeutet, dass das Window keine weiteren Daten mehr erwartet und die Berechnungen durchgeführt werden kann.

Wenn mit unendlichen Daten gearbeitet wird, für welche ein Fixed-Window von 5 Minuten definiert ist, werden für jedes Fenster alle Daten mit einem Zeitstempel innerhalb des angegebenen Zeitfensters gesammelt (z. B. zwischen 17:05:00 und 17:09:59 im ersten Fenster). Daten, welche einen Zeitstempel außerhalb dieses Bereichs aufweisen (Daten ab 17:10:00 Uhr oder später), gehören zu einem anderen Fenster.[8]

Da nicht gewährleistet werden kann, dass die Daten in einer Pipeline in einer zeitlichen Reihenfolge oder in vorhersehbaren Abständen eintreffen, wird ein Wasserzeichen(Watermark) verfolgt, welches dem System vorgibt, wann alle Daten in einem bestimmten Fenster in der Pipeline angekommen sein sollten. Sobald das Wasserzeichen über das Ende eines Fensters hinausgeht, wird jedes weitere Element, das mit einem Zeitstempel in diesem Fenster ankommt, als verspätetes Ereignis betrachtet.[8]

Wenn in dem Beispiel ein einfaches Watermark definiert wird, welches von einer Verzögerung von etwa 30 Sekunden zwischen den Zeitstempeln der Daten (der Event-Time) und dem Zeitpunkt des Erscheinens der Daten in der Pipeline (der Processing-Time) ausgeht, dann schließt das erste Fenster um 17:10:29 Uhr. Wenn ein Datensatz um 17:10:34 Uhr eintrifft, aber einen Zeitstempel hat, der in das Fenster von 0:00-4:59 Uhr eingeordnet werden würde (z. B. 17:05:38 Uhr), dann handelt es sich um ein verspätetes Ereignis, welches nicht in die Berechnung einfließt und verworfen wird.[8]

Es gibt zwei verschiedene Arten von Watermarks:

- Perfekte Watermarks
- Heuristische Watermarks

Bei den perfekten Watermarks gibt es keine verspäteten Daten, alle Ereignisse mit einem Ereigniszeitpunkt, welcher kleiner ist als das Watermark wurden beobachtet. Heuristische Watermarks können verspätete Daten enthalten, sie sind jedoch so genau wie möglich. Die Daten können verspätet sein, wenn der Abstand zwischen Verarbeitungszeit und Ereigniszeit zu groß ist.

Bei Watermarks sollte man aufpassen, dass diese die Verarbeitung nicht verlangsamen oder das viele Ereignisse übersehen werden.[2]

2.6 Beziehung der Konzepte

Um die Beziehungen zwischen all diesen Konzepten zu verstehen, werden vier Fragen beantwortet, die für jedes Problem der unbegrenzten Datenverarbeitung entscheidend sind:

Welche Ergebnisse werden berechnet? Dies wird durch die Arten der Transformationen innerhalb der Pipeline definiert. Dazu gehört z.B. das Berechnen von Summen, das Erstellen von Histogrammen oder auch das Trainieren von Modellen für maschinelles Lernen.[1]

Wo in der Ereigniszeit werden die Ergebnisse berechnet? Diese Frage wird durch die Verwendung von Windows innerhalb der Pipeline beantwortet.[1]

Wann werden die Ergebnisse in der Verarbeitungszeit materialisiert? Diese Frage wird durch die Verwendung von Wasserzeichen(Watermark) und Triggern beantwortet. Durch

Wasserzeichen wird abgegrenzt, wann die Eingabe für ein bestimmtes Fenster vollständig ist. Trigger hingegen ermöglichen die Ausgabe von frühen und spekulativen Teilergebnissen oder auch verspäteten Ergebnissen.[1]

Wie hängen die Verfeinerungen der Ergebnisse zusammen? Es kann in vielen Event-Streaming-Verarbeitungssystemen angegeben werden, ob die Ergebnisse akkumuliert werden sollen oder jedes Window ein einzelnes Ergebnis widerspiegelt.[1]

3 Event-Streaming-Verarbeitungssysteme

In diesem Kapitel werden die verschiedenen Event-Streaming-Verarbeitungssysteme, welche im Experiment evaluiert werden sollen, vorgestellt.

3.1 Apache Spark

Das Projekt Apache Spark wurde 2009 an der Universität Berkeley in Kalifornien gestartet. Das Projekt hatte das Ziel, eine einheitlichen Engine für die verteilte Datenverarbeitung zu entwickeln. Spark hat ein ähnliches Programmiermodell wie MapReduce.[22] Spark wurde um eine Abstraktion erweitert, welche für die Nutzung von Daten implementiert wurde. Diese Abstraktion wird "Resilient Distributed Datasets"(RDDs) genannt.[57] Mit dieser einfachen Erweiterung kann Spark eine breite Palette von Verarbeitungslasten erfassen, für die zuvor separate Engines erforderlich waren einschließlich SQL, Streaming, maschinelles Lernen und der Graphenverarbeitung. Seit der Veröffentlichung im Jahr 2010 hat sich Spark zum aktivsten Open-Source-Projekt im Bereich der Big-Data-Verarbeitung mit mehr als 1.000 Mitwirkenden entwickelt.[58] Spark bietet integrierte APIs in Java, Scala oder Python. Daher können die Anwendungen in verschiedenen Programmiersprachen implementiert werden.[55]

Innerhalb von Apache Spark gibt es zwei verschiedene Abstraktionen, um Daten zu verarbeiten. Es gibt Spark Streaming und Spark Structured Streaming.

Spark Streaming ist eine Abstraktion, welche den Kern von Spark erweitert, um kontinuierliche Daten zu verarbeiten. Die empfangenen Daten werden in Spark Streaming in sogenannte Mikro Batches unterteilt, bevor diese verarbeitet werden. Ein Mikro Batch ist eine kleine Sammlung von Daten. Spark Streaming verwendet als Datenabstraktion die 'Discretized Streams'.

Apache Spark bietet eine Garantie für eine 'exactly-once' Zustellung der Ereignisse.[30] 'Exactly-once' bedeutet, dass genau eine Übergabe des Ereignisses an den Empfänger erfolgt. Die Nachricht kann weder verloren gehen noch dupliziert werden.

Spark Structured Streaming hingegen verwendet die Dataframe- oder Dataset-APIs. Die Dataframe/Dataset-API ist optimierter und bietet eine bessere Entwicklungsunterstützung als RDDs wie SQL-Funktionen. Es ist einfach, ineffiziente Transformationen mit RDDs zu erstellen. Die Abstraktion DataFrame verwendet hingegen einen Catalyst-Optimierer, der einen Abfrageplan erstellt, welcher zu einer besseren Leistung führt. Der Catalyst-Optimierer ist eine wichtige Komponente von Apache Spark. Er optimiert strukturelle Abfragen, welche in SQL oder über die DataFrame/Dataset-APIs ausgedrückt werden.[18]

Spark verfügt über mehrere zentrale Abstraktionen: DataSets, DataFrames, SQL-Tabellen und Resilient Distributed Datasets (RDDs). Diese Abstraktionen haben unterschiedliche Schnittstellen, um mit den Datenabstraktionen zu arbeiten. Am einfachsten und effizientesten sind DataFrames, die in allen Programmiersprachen verfügbar sind.[14] Zusätzlich zu den verschiedenen Abstraktionen machen Systeme wie Spark Streaming Annahmen darüber, wie viele Daten in den Speicher passen und wie schnell sich die Last ändert.[47]

Das Hauptmerkmal von Spark ist das 'In-Memory-Cluster-Computing', das die Verarbeitungsgeschwindigkeit einer Anwendung erhöht. Spark ist für ein breites Spektrum von Anwendungen für die parallele Datenverarbeitung konzipiert wie Batch-Anwendungen, interaktive Abfragen und Streaming.[55]

3.1.1 Resilient Distributed Datasets

Die wichtigste Datenabstraktionen in Apache Spark sind die Resilient Distributed Datasets(RDDs). RDDs sind eine Abstraktion von einem verteilten Speicher, die es Programmierern ermöglicht, parallele speicherinterne Berechnungen in großen Clustern auf einer fehlertoleranten Weise durchzuführen.

RDDs sind eine schreibgeschützte und partitionierte Sammlung von Datensätzen. Spark stellt RDDs über ein funktionales Programmier-Interface in Scala, Java, Python und R bereit. Benutzer erstellen RDDs indem deterministische Operationen, die als Transformationen bezeichnet werden (z.B. map, filter oder groupBy), auf Daten im Speicher oder anderen RDDs angewendet werden.

Anschließend können mit diesen RDDs Aktionen ausgeführt werden, z.B. Operationen, die einen Wert an die Anwendung zurückgeben oder Daten in ein externes Speichersystem zu exportieren. So ist es möglich, auf einem RDD die Aktion “count” auszuführen, welche berechnet, wie viele Elemente im Datensatz vorhanden sind.

RDDs werden bei der ersten Berechnung “lazy” evaluiert. Spark kann so einen effizienten Plan für die Berechnung des Benutzers finden. Erst wenn eine Aktion aufgerufen wird, betrachtet Spark den gesamten Graphen der Transformationen, die zur Erstellung eines Ausführungsplans verwendet werden. Wenn es beispielsweise mehrere Filteroperationen in einer Reihe gibt, kann Spark diese in einem Durchgang zusammenfassen. Hierdurch wird die Performance optimiert.[57]

Es ist zudem möglich, eine “"Persist-Methode“ aufzurufen, um anzugeben, welche RDDs in zukünftigen Operationen wiederverwendet werden. Spark behält persistente RDDs standardmäßig im Speicher, diese können aber auch auf die Festplatte ausgelagert werden, wenn nicht genügend RAM vorhanden ist. Benutzer können auch andere Persistenzstrategien deklarieren, wie das Speichern des RDD nur auf der Festplatte oder das Replizieren über Maschinen hinweg. Schließlich können Benutzer eine Persistenzpriorität für jedes RDD festlegen, um anzugeben, welche In-Memory-Daten zuerst auf die Festplatte ausgelagert werden sollen. Hierdurch können wichtige Daten im Arbeitsspeicher gehalten werden, während unwichtigere Daten ausgelagert werden.[57]

Resilient Distributed Dataset (RDD) besitzen die Eigenschaft fehlertolerant zu sein. Dies wird durch einen gerichtete azyklischen Graphen(DAG) erreicht, welcher im Hintergrund alle Änderungen und Aktivitäten steuert, die für die Aufgabe erforderlich sind. Falls in einer Anwendung ein Fehler auftritt oder ein Ausfall eintritt, können anhand des DAG die Daten wiederhergestellt werden. Falls eine Partition eines RDD verloren geht, verfügt das RDD über genügend Informationen darüber, wie es von anderen RDDs abgeleitet wurde, um die Daten neu zu berechnen. So können verlorene Daten recht schnell wiederhergestellt werden, ohne dass eine kostspielige Replikation erforderlich ist.[57]

RDDs sind am besten für Batch Anwendungen geeignet, die denselben Vorgang auf allen Elemente eines Datensatzes anwenden. In diesen Fällen können RDDs sich jede Transformation effizient als einen Schritt in einem “lineage graph” merken und können verlorene Partitionen wiederherstellen, ohne große Datenmengen protokollieren zu müssen. RDDs sind weniger geeignet für Anwendungen, die asynchrone Aktualisierungen des gemeinsamen Zustands vornehmen, wie ein Speichersystem für eine Webanwendung oder einen inkrementellen Webcrawler. Für diese Anwendungen ist es effizienter, Systeme zu

verwenden, die herkömmliche Aktualisierungsprotokollierung und Daten-Checkpointing durchführen wie Datenbanken.[57]

Wann sollte man RDDs nutzen:

- Batchverarbeitung
- Low-level Transformationen, Aktionen und Kontrolle über den Datensatz.
- Unstrukturierte Daten, wie Media Streams oder einen Text Stream.
- Kein Schema vorhanden sein muss.
- Verzichten auf Optimierungs- und Leistungsvorteile, die mit der DataFrames und Datasets API für strukturierte und halbstrukturierte Daten verfügbar sind

[21]

3.1.2 Dataframe/DataSet

Eine weitere Datenabstraktion in Apache Spark ist der Dataframe. Wie ein RDD ist ein DataFrame eine unveränderliche und verteilte Sammlung von Daten.[21] Im Gegensatz zu einem RDD sind die Daten in benannten Spalten organisiert wie eine Tabelle in einer relationalen Datenbank.[49] Der DataFrame wurde entwickelt, um die Verarbeitung großer Datensätze zu vereinfachen. Der Dataframe ermöglicht es Entwicklern, einer verteilten Datensammlung eine Struktur zu geben, die eine Abstraktion auf einer höheren Ebene ermöglicht. Die Informationen aus der Struktur der Daten können benutzt werden, um extra Optimierungen durchzuführen.[49] Es bietet ein Interface, um die verteilten Daten zu manipulieren. Seit Apache Spark 2.0 wurde die DataFrame-API mit der Dataset-API integriert. Hierdurch wurden die Datenverarbeitungsfunktionen in allen Bibliotheken vereinheitlicht. Aufgrund dieser Vereinheitlichung müssen Entwickler nun weniger Konzepte erlernen und arbeiten mit einer einzigen hochrangigen und typsicheren API namens Dataset.

Die Spark-Kern-API basierte auf funktionaler Programmierung über verteilte Sammlungen, die beliebige Typen von Scala-, Java- oder Python-Objekten enthalten. Dieser Ansatz ist zwar sehr ausdrucksstark, erschwerte aber auch die automatische Analyse

und Optimierung von Programmen. Die in RDDs gespeicherten Scala-/Java-/Python-Objekte können eine komplexe Struktur aufweisen und die über sie ausgeführten Funktionen können beliebigen Code enthalten. In vielen Anwendungen erzielen Entwickler eine suboptimale Leistung, wenn nicht die richtigen Operatoren verwendeten werden.[58]

Um dieses Problem zu lösen, wurde Spark 2015 um eine deklarative API namens DataFrames erweitert, die auf relationaler Algebra basiert. DataFrames sind eine gängige Schnittstelle für tabellarische Daten in Python und R. Ein DataFrame ist ein Datensatz mit einem bekannten Schema, der im Wesentlichen einer Datenbanktabelle ähnelt und Operationen wie Filterung und Aggregation unterstützt.[58] Im Gegensatz zur SQL-Sprache werden DataFrame Operationen als Funktionsaufrufe in einer allgemeineren Programmiersprache wie Python, Java oder R aufgerufen, sodass Entwickler die Anwendungen mithilfe von Abstraktionen leicht strukturieren und implementieren können. [58]

Die Berechnungen beim Dataframe werden parallelisiert und optimiert. Dies geschieht automatisch mit der Hilfe des Abfrageplaners von Spark SQL. Der Benutzercode erhält Optimierungen wie Prädikat-Pushdown, die unter der funktionalen API von Spark nicht verfügbar sind. Prädikat-Pushdown ist eine Optimierung, um Filterbedingungen so früh wie möglich anzuwenden.[58]

DataFrames haben sich schnell zu einer beliebten Schnittstelle entwickelt.[58] In einer Umfrage vom Juli 2015 gaben 60% der Befragten an, sie zu verwenden.[58] Aufgrund des Erfolgs von DataFrames wurde eine typsichere Schnittstelle namens Datasets entwickelt, mit der Java- und Scala-Programmierer DataFrames als statisch typisierte Sammlungen von Java-Objekten betrachten können. Diese API wird allmählich zur Standardabstraktion für die Datenübergabe zwischen den Spark-Bibliotheken.[58]

3.1.3 Discretized Streams

Die Datenabstraktion der Spark Streaming Schnittstelle wird als diskreter Stream (D-Stream) bezeichnet, der als eine Reihe von kurzen, zustandslosen, deterministischen Aufgaben definiert ist. Intern wird ein D-Stream durch eine fortlaufende Reihe von RDDs dargestellt, Sparks Abstraktion eines unveränderlichen, fehlertoleranten und verteilten Datensatzes. In Spark Streaming wird die Streaming-Berechnung als eine Reihe von deterministischen Batch-Berechnungen in kleinen Zeitintervallen behandelt. Ähnlich wie bei MapReduce ist ein Job in Spark als eine parallele Berechnung definiert, die aus mehreren Aufgaben besteht. Eine Aufgabe ist eine Arbeitseinheit, die an den Task-Manager

gesendet wird. Wenn ein Stream in Spark eintritt, werden die Daten in Mikro-Batches unterteilt, die die Eingabedaten der Resilient Distributed Datasets (RDD) sind. Anschließend werden Datenumwandlungen an diesen RDDs vorgenommen, die wiederum einen D-Stream ausgeben. Die unendlichen und kontinuierlichen Daten werden als Mikro-Batches verarbeitet.[45]

3.2 Flink

Apache Flink ist ein Open-Source-Framework für die verteilte Stream- und Batch-Datenverarbeitung. Es konzentriert sich auf die Verarbeitung großer Datenmengen mit sehr geringer Datenlatenz und hoher Fehlertoleranz auf verteilten Systemen. Das Hauptmerkmal von Flink ist die Fähigkeit, unendliche und konitinuierlichen Daten in Echtzeit zu verarbeiten.[27]

Der Kern von Flink ist die verteilte Engine, welche die Streaming-Anwendungen ausführt. Ein Flink-Laufzeitprogramm ist ein gerichteter azyklischer Graph(DAG) von zustandsabhängigen Operatoren. Es existieren zwei Kern-APIs in Flink: die DataSet-API für die Verarbeitung endlicher Datensätze (Batchverarbeitung) und die DataStream-API für die Verarbeitung potenziell unbegrenzter Datenströme (Stream-Verarbeitung). Die DataSet- als auch die DataStream-APIs erstellen Datenfluss-Programme, die von der Flink-Engine ausgeführt werden können. Als solche dient diese als gemeinsame Struktur zur Abstraktion von begrenzter (Batch) und unbegrenzter (Stream) Datenverarbeitung.

Flink bündelt domänenspezifische Bibliotheken und APIs, welche die DataSet- und DataStream Programme erzeugen. Es existiert derzeit FlinkML für maschinelles Lernen, Gelly für die Graphenverarbeitung und Table für SQL-ähnliche Operationen. Diese sind über der Kern API angeordnet.[15] Siehe Abbildung:3.1

Alle Flink-Programme werden verzögert ausgeführt. Wenn die Hauptmethode des Programms ausgeführt wird, werden die Daten nicht direkt geladen und transformiert. Vielmehr wird jede Operation erstellt und dem Programmplan hinzugefügt. Die Operationen werden erst ausgeführt, wenn die Ausführung explizit durch einen execute()-Aufruf in der Ausführungsumgebung ausgelöst wird. Mit der ‘’lazy’ Evaluation können anspruchsvolle Programme konstruiert werden, die von Flink als eine ganzheitlich geplante Einheit ausgeführt werden.[24]

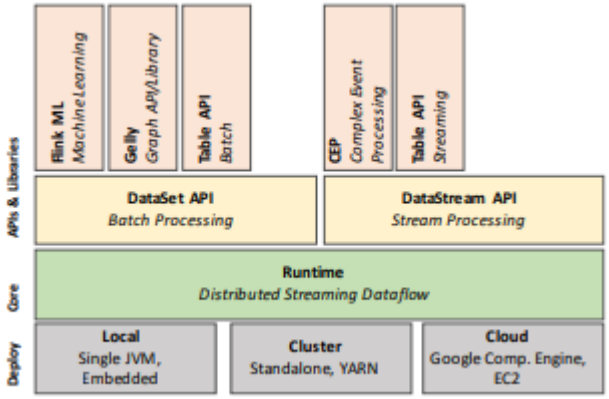

Abbildung 3.1: Flink APIs[15]

Flink bietet eine zuverlässige Ausführung mit strikten Konsistenzgarantien für eine 'exactly-once' Verarbeitung an und behandelt Fehler, die im System auftreten können, durch Checkpointing und einer teilweisen Neuausführung des DAG. Zur effektiven Bereitstellung dieser Garantien wird die allgemeine Annahme angestellt, dass die Datenquellen persistent und wiederholbar ist. Beispiele für solche Quellen sind Dateien und langlebige Message-Broker wie Apache Kafka.[15]

3.2.1 DataStream

Ein DataStream ist eine unveränderliche und unbegrenzte Sammlung von Daten, welche Duplikate enthalten kann. Diese Sammlung unterscheidet sich von normalen Java-Sammlungen in einigen wichtigen Punkten. Erstens sind sie unveränderlich, d.h. es können keine Elemente nach der Erstellung hinzugefügt oder entfernt werden. Außerdem können die darin enthaltenen Elemente nicht einfach inspiziert werden, sondern nur mit den Operationen der DataStream-API bearbeiten werden.[24] Eine Sammlung wird zunächst durch das Hinzufügen einer Quelle in einem Flink-Programm erstellt. Neue Sammlungen werden von diesen abgeleitet, indem die API-Methoden wie map, filter usw. angewendet und die Daten transformiert werden.

Der DataStream ist eine Sequenz von teilweise geordneten Datensätzen. DataStreams sind vergleichbar mit Storm-Tupeln. Die DataStreams empfangen kontinuierliche Daten aus externen Quellen wie Messsage-Broker oder Sockets. Die DataStreams-API unterstützt mehrere Operationen und Funktionen wie der Filterung und der Reduktion,

die inkrementell auf jedes Ereignis angewendet werden und einen neuen DataStream erzeugen. Jede Operationen oder jede Funktion kann parallelisiert werden, indem Instanzen auf verschiedenen Partitionen des jeweiligen Streams ausgeführt werden. Diese Methode ermöglicht eine verteilte Ausführung auf den kontinuierlichen und unbegrenzten Daten.[45]

3.2.2 DataSet

Ein DataSet ist eine unveränderliche und begrenzte Sammlung von Daten, die Duplikate enthalten können. Hier gilt das Gleiche wie bei der DataStream API. Siehe:3.2.1 DataSets werden aus Quellen wie lokalen Dateien oder durch Lesen einer Datei aus einer bestimmten Quelle erstellt. Anschließend können die Funktionen wie Map oder Filter auf den Daten angewendet werden. Die Ergebnisse können in verschiedene 'Sinks', zum Beispiel Dateien oder Kommandozeilenterminals geschrieben werden. Diese Schnittstelle wird für die Verarbeitung von Batch-Daten verwendet.[24]

3.2.3 Table

Apache Flink bietet eine Tabellen-API als einheitliche und relationale API für die Batch- und Stream-Verarbeitung an. Abfragen werden mit derselben Semantik auf unbegrenzten Echtzeit-Streams oder begrenzten Batch-Datensätzen ausgeführt und liefern dieselben Ergebnisse. Die Tabellen-API in Flink wird häufig verwendet, um schnelle Datenanalysen und ETL-Anwendungen zu erleichtern.

Die Tabellen-API folgt dem relationalen Modell: Tabellen haben ein zugehöriges Schema und die API bietet vergleichbare Operationen wie Select, Join, Group-by oder Aggregate.[25] Tabellen-API-Programme legen deklarativ fest, welche logische Operation ausgeführt werden soll, anstatt genau zu spezifizieren, wie der Code für die Operation aussieht. Obwohl die Tabellen-API durch verschiedene Arten von benutzerdefinierten Funktionen erweiterbar ist, ist sie weniger aussagekräftig als die Kern-APIs, dafür aber prägnanter in der Anwendung. Darüber hinaus durchlaufen die Programme der Tabellen-API einen 'Optimizer', der vor der Ausführung Optimierungsregeln anwendet. Hierdurch werden die Abfragen performanter.[24]

SQL

Die höchste von Flink angebotene Abstraktionsebene ist SQL. Diese Abstraktion ähnelt der Tabellen-API sowohl in der Semantik als auch in der Ausdruckskraft.[25] Die SQL-Abstraktion interagiert eng mit der Tabellen-API. SQL-Abfragen können über Tabellen ausgeführt werden, welche in der Tabellen-API definiert sind.[24]

3.3 Apache Storm

Apache Storm wurde in Java und Clojure implementiert und ist ein freies verteiltes Event-Stream-Verarbeitungssystem für Echtzeitberechnungen. Apache Storm ist unter der Apache-Lizenz lizenziert. Es beinhaltet Fehlertoleranz, Skalierbarkeit und Garantien für die Datenverarbeitung. Storm verfügt über verschiedene Stufen der garantierten Nachrichtenverarbeitung einschließlich 'best effort', 'at-least-once' und 'exactly-once' durch Trident. 'Best-effort' beschreibt, dass es keine Garantie dafür bietet, dass die Daten zugestellt werden oder dass die Zustellung einer bestimmten Dienstqualität entspricht. 'At-least-once' bedeutet, dass Nachrichten dupliziert werden können, aber nicht verloren gehen. Es können potenziell mehrere Zustellungsversuche unternommen werden, von denen mindestens einer erfolgreich ist.

Apache Storm eignet sich hervorragend gut für die Verarbeitung von unbegrenzten Datenströmen geeignet.

In Storm existieren drei übergeordnete Abstraktionen: Spouts, Bolts und Topologien. Ein Spout ist eine Quelle von Streams. Die Abstraktion von Streaming Daten werden als Tupel bezeichnet, die aus den Daten und einem Bezeichner bestehen. In Storm bestehen Anwendungen aus Topologien, die einen gerichteten azyklischen Graphen bilden, der aus Eingabeknoten, den sogenannten Spouts, und Verarbeitungsknoten, den Bolts, besteht. Ein Bolt kapselt eine logische Berechnung, die eine beliebige Anzahl von Eingabeströmen verarbeitet und eine beliebige Anzahl von neuen Ausgabeströmen erzeugt. Ein Spout ist eine Quelle von Streams in einer Topologie. Im Allgemeinen lesen Spouts Tupel aus einer externen Quelle und geben diese weiter. Eine Topologie ist also ein Netzwerk aus Spouts und Bolts, wobei jede Kante des Netzwerks einen Bolt darstellt, der den Ausgangsstrom eines anderen Spouts oder Bolts abonniert.[19][45]

Apache Storm ist fehlertolerant. Der Cluster in Storm besteht aus einem Nimbus und einem Supervisor. Der Nimbus-Knoten ist der Master in einem Storm-Cluster. Dieser ist verantwortlich für die Verteilung des Anwendungscodes auf den verschiedenen Arbeitsknoten, die Zuweisung von Aufgaben an die verschiedenen Maschinen, die Überwachung von Aufgaben und Ausfällen und deren Neustart bei Bedarf.[34] Der Cluster behält einen Überblick über alle Worker, wenn ein Knoten stirbt, startet Apache Storm den Prozess auf einem anderen Knoten neu. Der Nimbus und die Supervisors sind zustandslos und ausfallsicher. Wenn diese ausfallen, werden sie neu gestartet, als wäre nichts geschehen. Dies ist ein Grund, weswegen Fehler auftreten können, ohne dass es zu einem katastrophalen Ausfall des gesamten Storm-Clusters kommt.[33]

3.3.1 Tupel

Das Tupel ist die wichtigste Datenstruktur in Storm. Die Knoten in einer Topologie senden untereinander Daten in Form von Tupeln. Ein Tupel ist eine benannte Liste von Werten, wobei jeder Wert von beliebigem Typ sein kann.[54] Der Wert jedes Feldes kann ein Byte, Char, Integer, Long, Float, Double, Boolean oder ein Byte-Array sein. Ein Tupel ist dynamisch typisiert d.h. es müssen nur die Namen der Felder in einem Tupel definiert werden, nicht aber deren Datentypen. Die Wahl der dynamischen Typisierung trägt zur Vereinfachung der Schnittstelle bei und erleichtert die Verwendung.[34] Storm muss jedoch wissen, wie diese Werte zu serialisieren sind, damit es das Tupel zwischen den Knoten in der Topologie senden kann. Apache Storm hat standardmäßig Serialisierer für die primitiven Typen implementiert, für spezifische Typen müssen benutzerdefinierte Serialisierer implementiert und registiert werden. Tupels haben Hilfsmethoden wie getInteger und getString, um Feldwerte zu erhalten, ohne dass das Ergebnis konvertiert werden muss. Tupeln werden mithilfe einer Funktion an die nächsten Knoten in der Topologie gesendet.[5]

3.3.2 Stream

Die zentrale Abstraktion in Storm ist der SStream". Ein Stream ist eine unbegrenzte Folge von Tupeln, die von Storm parallel verarbeitet werden können.[34] Jeder Stream kann von einem einzelnen oder mehreren Typen von Bolts(Siehe:3.3.4) verarbeitet werden. Allen Streams in einer Storm-Anwendung werden IDs zugewiesen und die Bolts können Tupel aus diesen Streams auf der Grundlage der ID produzieren und konsumieren.[34]

Eine Topologie(Siehe:3.3.5) kann eine beliebige Anzahl von Streams enthalten. Mit Ausnahme des allerersten Knotens in der Topologie, welcher als Dateneingabe dient. Die Knoten führen Berechnung oder Transformation anhand der Eingabetupel durch und geben neue Tupel aus, wodurch ein neuer 'Output-Stream' entsteht. Diese 'Output-Streams' dienen dann als 'Input-Streams' für weitere Knoten.[5]

3.3.3 Spout

Spout ist die Abstraktion in Storm, welche Daten aus einer Quelle lesen und diese an die übergeordnete Topologie als Tupel weitergeben.[34] Ein Spout kann sich zum Beispiel mit der Twitter-API verbinden und einen Stream von Tweets an die übergeordnete Topologie ausgeben. Spouts können entweder zuverlässig oder unzuverlässig sein. Ein zuverlässiger Spout ist in der Lage, ein Tupel wiederzugeben, wenn es von Storm nicht verarbeitet werden konnte, während ein unzuverlässiger Spout das Tupel vergisst, sobald es gesendet wurde.

Spouts werden mit dem IRichSpout Interface implementiert. Hierfür müssen verschiedene Funktionen implementiert werden. Die wichtigsten Funktionen sind nextTuple(), ack() und fail().[54] Die funktion nextTuple sendet entweder ein neues Tupel in die Topologie oder kehrt zurück, wenn es keine neuen Tupel zu senden gibt. Es ist unbedingt erforderlich, dass nextTuple bei keiner Spout-Implementierung blockiert, da Storm alle Spout-Methoden auf demselben Thread aufruft. Die anderen Hauptmethoden für Spouts sind ack und fail. Sie werden aufgerufen, wenn Storm feststellt, dass ein vom Spout gesendetes Tupel entweder erfolgreich die Topologie durchlaufen hat oder nicht abgeschlossen werden konnte. Ack und fail werden nur für zuverlässige Spouts aufgerufen.[5]

3.3.4 Bolt

Eine weitere Abstraktion in Apache Storm ist der Bolt. Ein Bolt ist für die gesamte Verarbeitung in Topologien zuständig. Bolts können Daten filtern, aggregieren, joins durchführen und zum Beispiel mit Datenbanken kommunizieren. Ein Bolt wird für einfache Stream-Transformationen benutzt.[34] Für komplexe Stream-Transformationen sind oft mehrere Schritte und damit mehrere Bolts erforderlich. Die Umwandlung eines Streams von Daten in eine Gruppierung mit Aggregation erfordert beispielsweise mindestens in zwei Schritten: Einen Bolt, der eine fortlaufende Gruppierung und Aggregation der

Daten vornimmt sowie einen oder mehrere Bolts, die die verarbeiteten Daten als Stream ausgeben.

Bolts werden mit dem IRichBolt Interface implementiert. Es existiert zudem noch ein weiteres Interface für Bolts, das IBasicBolt Interface, welches für simple Filterungen oder Funktionen geeignet ist. Bei beiden Interfaces müssen dieselben Funktionen implementiert werden.[54]

Die wichtigste Methode in Bolts ist die Execute-Methode, die als Eingabe ein neues Tupel erhält. Nachdem das Tupel verarbeitet worden ist, wird das neue Tupel mithilfe des OutputCollector-Objekts ausgegeben. Bolts müssen für jedes Tupel, das sie verarbeiten, die ack-Methode auf dem OutputCollector aufrufen, damit Storm weiß, wann die Verarbeitung des Tupel abgeschlossen ist. Es ist zudem möglich, Threads in Bolts zu starten, die die Verarbeitung asynchron durchführen, da der OutputCollector threadsicher ist und jederzeit aufgerufen werden kann.[5]

3.3.5 Topology

Die Logik für eine Echtzeitanwendung wird in eine Storm-Topologie verpackt. Eine Topologie besteht aus Spouts, Bolts, Streams und Tupeln.[34] Diese werden in eine 'Topologie' verpackt, die die oberste Abstraktionsebene darstellt und dem Storm-Cluster zur Ausführung übergeben wird. Eine Topologie kann durch einen direkten azyklischen Graphen(DAG) dargestellt werden, bei dem jeder Knoten eine Art von Verarbeitung durchführt und diese an den oder die nächsten Knoten weiterleitet. [34] Die Kanten im Graphen zeigen an, welche Bolts welche Streams abonniert haben.(Siehe:3.2) Wenn ein Spout oder Bolt ein Tupel an einen Stream sendet, sendet er das Tupel an jeden Bolt, der diesen Stream abonniert hat. Dies ist auch in der folgenden Abbildung gut zu erkennen.(Siehe:3.2)

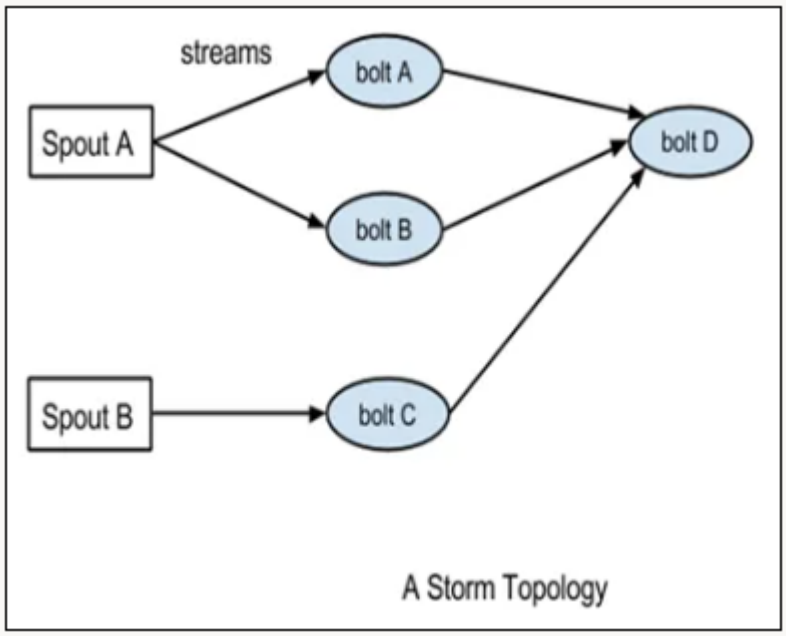

Abbildung 3.2: Storm Topology[34]

3.3.6 Trident

Trident ist eine High-Level-Abstraktion für die Durchführung von Echtzeitberechnungen auf der Grundlage von Storm. Der Hauptvorteil bei der Verwendung von der Trident Abstraktion besteht darin, dass jede in die Topologie eingegebene Nachricht nur einmal verarbeitet wird.[34]

3.4 KSQLDB

KsqlDB ist eine Open-Source-Event-Streaming-Datenbank, die von Confluent 2017 veröffentlicht wurde und unter der Confluent Community License Agreement lizenziert. Das System vereinfacht die Art und Weise, wie Stream-Verarbeitungs-Anwendungen erstellt, bereitgestellt und gewartet werden.[50]

KsqlDB ist ein dezentrales System und eine Event-Streaming-Datenbank, welche die Speicherung und Verarbeitung von kontinuierlichen Daten in SQL ermöglicht.

KsqlDB dient der Entwicklung von Stream-Verarbeitungs-Anwendungen auf der Grundlage von Apache Kafka. Angetrieben von der Kafka Streams API ist KsqlDB ein robustes Stream-Verarbeitungssystem, welches einem nur mit SQL ermöglicht, eine standardmäßige Stream-Verarbeitungs-Anwendungen zu erstellen. KsqlDB erweitert die Kafka Streams API um weitere Funktionen wie einen Streaming-Server und eine einfach zu bedienende SQL-Schnittstelle. Es ist ein verteiltes, skalierbares Stream-Verarbeitungs-Framework, welches Daten in 'Echtzeit' verarbeitet.[50] KsqlDB unterstützt zwei verschiedene Verarbeitungsgarantien 'at-least-once' und 'exactly-once', welche in der Konfiguration von KsqlDB deklariert werden können.[43]

KsqlDB verwendet eine lokale Festplatte, um den temporären Zustand für Aggregationen und Joins zu erhalten.[42] Während die zugrunde liegende Speicherung Kafka-Themen sind, stellt KsqlDB die Daten in diesen Themen als Streams oder Tabellen dar.

Die beiden primären Abstraktionen in KsqlDB sind Streams und Tabellen.

3.4.1 Stream

Ein Stream ist eine partitionierte, unveränderliche, "append-only" Sammlung, die eine Reihe von historischen Fakten darstellt. Zeilen in einem Stream sind unveränderlich. Wird ein Event in einem Stream eingefügt, kann dieses nicht aktualisiert werden. Neue Zeilen können am Ende des Streams angehängt werden, aber vorhandene Zeilen können nicht aktualisiert oder gelöscht werden. Jedes Event wird in einer bestimmten Partition gespeichert und besitzt implizit oder explizit einen Schlüssel, der ihre Identität darstellt. Alle Events mit dem gleichen Schlüssel befinden sich auf derselben Partition, dadurch erfolgt ein schnellerer Zugriff bei Gruppierungen, die als Gruppierungsvariable den Schlüssel benutzen.[43]

3.4.2 Table

Eine Tabelle ist eine veränderliche, partitionierte Sammlung, die Veränderungen im Laufe der Zeit modelliert. Im Gegensatz zu einem Stream, der eine historische Abfolge von Ereignissen darstellt, stellt eine Tabelle dar, was im Moment wahr ist. Tabellen funktionieren, indem sie die Schlüssel der einzelnen Zeilen nutzen. Wenn eine Folge von Ereignissen einen gemeinsamen Schlüssel hat, stellt das letzte Ereignis für einen bestimmten Schlüssel die aktuellsten Informationen dar. Im Hintergrund wird ein Prozess angestoßen, der

regelmäßig ausgeführt wird und alle Zeilen löscht außer die neueste für jeden Schlüssel. KsqlDB verwendet Kafkas Konzept eines kompaktierten Themas. Die Kompaktierung ist ein Prozess, der in regelmäßigen Abständen alle Ereignisse außer das neueste für jeden Schlüssel löscht.[43]

3.5 Kafka Streams

Kafka Streams gehört zu einer Gruppe von Technologien, die zusammenfassend als Kafka-Ökosystem bezeichnet werden. Apache Kafka ist ein verteiltes und 'append-only' Protokoll, in das Nachrichten eingespeist werden und aus dem Nachrichten ausgelesen werden können. Die Aufteilung des verteilten Protokolls in Partitionen ermöglicht es, dass das System skalieren kann. Darüber hinaus enthält Kafka einige wichtige APIs für die Interaktion mit den Daten.[50]

Die erste Version von Kafka Streams, wurde im Jahr 2016 veröffentlicht.[50] Kafka Streams ist auf die Verarbeitung von Echtzeit-Datenströmen ausgerichtet und nicht nur auf die Übertragung von Daten. Mit Kafka Streams ist es einfach kontinuierliche und unendliche Daten zu konsumieren, diese anzureichern, zu transformieren und zu verarbeiten.

Bevor es Kafka-Streams gab, existierten zwei Optionen, um für Kafka-basierte Stream-Verarbeituns-Anwendungen zu entwickeln: Die direkte Verwendung der 'Consumer- und Producer'-APIs oder die Verwendung eines anderes Stream-Verarbeitungssystem wie Apache Spark oder Apache Flink. Wenn etwas nicht triviales gemacht werden soll, wie Datensätze aggregieren, getrennte Streams zusammenführen, Ereignisse in Zeitfenstern zu gruppieren oder Ad-hoc-Abfragen durchzuführen, stößt man mit der direkten Verwendung der Producer und Consumer -API an seine Grenzen. Die Producer und Consumer-APIs enthalten keine Abstraktionen, die bei diesen Arten von Anwendungsfällen helfen. Die zweite Option ist die Einführung einer vollständigen Streaming-Plattform wie Apache Spark oder Apache Flink, dies führt jedoch zu einer enormen Komplexität.[50]

Kafka Streams baut auf den in Kafka nativ integrierten Funktionen zur Fehlertoleranz auf. Kafka-Partitionen sind hochverfügbar und repliziert. Wenn kontinuierliche Daten in Kafka persistiert werden, sind diese auch dann verfügbar, wenn die Anwendung ausfällt und diese erneut verarbeitet werden müssen.[36] Kafka Streams unterstützt 'at-least-once' und 'exactly-once' Verarbeitungsgarantien. Die Verarbeitungsgarantien können in der Konfiguration von Kafka-Streams deklariert werden.[20]

Eine verbreitete Vorstellung im Bereich der Softwareentwicklung ist, dass Abstraktionen in der Regel einen Preis haben: Je mehr man von den Details abstrahiert, desto mehr fühlt sich die Software wie 'magisch' an und desto mehr Kontrolle gibt man auf.[50] Wie hoch der Kontrollverlust ist, hängt bei Kafka Streams von der gewählten Abstraktionsebene ab. Kafka Streams erlaubt es den Entwicklern, die Abstraktionsebene zu wählen, die für den Entwickler am besten geeignet ist. Dies ist abhängig vom Anwendungsfall und der Erfahrung des Entwicklers.[50]

Es existieren zwei verschiedene APIs für die Verarbeitung von kontinuierlichen Daten in Kafka Streams.

- Eine High-Level-DSL, die wie die Streaming-API von Java aussieht und funktioniert. Die DSL bietet einen flüssigen und funktionalen Ansatz für die Verarbeitung von Datenströmen. Diese API ist leicht zu erlernen und zu verwenden.
- Eine Low-Level-Prozessor-API, die Entwicklern bei Bedarf eine feine Steuerung ermöglicht.

[50]

Die High-Level-DSL baut auf der Prozessor-API auf, aber die Schnittstelle, welche beide zur Verfügung stellen, ist unterschiedlich. Wenn die Stream-Verarbeitungs-Anwendung einen funktionalen Programmierstil verwendet und Abstraktionen auf einer höheren Ebene für die Arbeit mit den Daten (Streams und Tabellen) benutzt werden soll, dann ist die DSL-API die richtige Wahl. Der Vorteil der Verwendung der High-Level-DSL gegenüber der Lower-Level-Prozessor-API in Kafka Streams ist, dass die DSL-API eine Reihe von Abstraktionen enthält, die das Arbeiten mit Streams und Tabellen vereinfacht.[50]

Werden Zugriffe auf die Daten auf einer niedrigen Ebene benötigt, periodische Funktionen geplant oder eine feine Kontrolle über das Timing bestimmter Vorgänge benötigt, ist die Prozessor-API die bessere Wahl.[50]

Kafka Streams erlaubt es, sowohl die DSL als auch die Prozessor-API in einer Anwendung zu verwenden, sodass man sich nicht auf eine der beiden Optionen festlegen muss. Es kann die DSL-API für Standardoperationen verwenden werden und die Prozessor-API für komplexere oder einzigartige Funktionen, die einen untergeordneten Zugriff auf den Verarbeitungskontext, den Status oder die Meta-Daten ermöglicht.[50]

Bevor Daten benutzt werden können, muss entschieden werden, welche Abstraktion verwendet werden soll. Bei dieser Entscheidung sollte berücksichtigt werden, ob nur der

letzte Zustand bzw. die letzte Repräsentation eines bestimmten Schlüssels oder die gesamte Historie der Ereignisse verfolgt werden soll.[50]

3.5.1 Streams

Ein Stream liefert unveränderliche Daten und unterstützt nur das Einfügen neuer Ereignisse, während vorhandene Ereignisse nicht geändert werden können.[50] Streams sind beständig, dauerhaft und fehlertolerant. Ereignisse in einem Stream können mit Schlüsseln versehen werden. Zudem können viele Ereignisse denselben Schlüssel besitzen.

Der Stream ist die wichtigste bereitgestellte Abstraktion von Kafka Streams und stellt einen unbegrenzten, sich ständig aktualisierenden Datensatz dar. Ein Datensatz wird als Schlüssel-Wert-Paar(Key-Value-Pair) definiert und ein Stream ist eine geordnete und wiederholbare Folge von unveränderlichen Datensätzen.

Ein Key-Value-Pair ist ein Datensatz, der zwei verbundene Gruppen durch einen Schlüssel und einen Wert darstellt. In Key-Value-Pairs können große Datenmengen gespeichert werden, sodass Programmierer Informationen leichter organisieren und sortieren können.

KStream

Einer der Vorteile der Verwendung der High-Level-DSL gegenüber der Lower-Level-Prozessor-API in Kafka Streams ist, dass die DSL-API eine Reihe von Abstraktionen enthält, die das Arbeiten mit Streams und Tabellen vereinfacht. Darunter zählt die Abstraktion KStream, KTable und GlobalKTable.[50]

Ein KStream ist eine Abstraktion von partitionierten und unendlichen Daten in der DSL-API, welche aus Key-Value Paaren besteht. Dabei stellt jeder Datensatz eine in sich geschlossene Dateneinheit in der unbeschränkten Datenmenge dar. Datensätze in einem Stream werden immer als 'Einfügen' interpretiert. Die vorhandenen Datensätze werden nicht durch neue Datensätze mit dem gleichen Schlüssel ersetzt. Dieser Ansatz wird häufig bei Kreditkartentransaktionen, Seitenaufruf-Ereignissen oder Server-Protokolleinträgen angewendet.[35] In einem KStream werden die Daten semantisch dargestellt, jedes Ereignis wird unabhängig von den anderen Ereignissen eingestuft. Wenn die folgenden 2

Datensätze: ("HAW", 100) -> ("HAW", 200) gestreamt werden und die Werte pro Benutzer summiert werden, wird 300 für “HAW” zurückgegeben. Dies liegt daran, dass der zweite Datensatz nicht als Aktualisierung des vorherigen Datensatzes betrachtet wird.

3.5.2 KGroupedStream

KGroupedStream ist eine Abstraktion eines gruppierten Streams von Key-Value-Pairs. Es handelt sich um eine Zwischendarstellung nach einer Gruppierung eines KStream. Anschließend kann eine Aggregation auf die gruppierten und partitionierten Daten angewendet werden. Nach der Aggregation erhält man eine KTable.[35]

3.5.3 Table

Eine Tabelle enthält veränderbare Daten. Neue Ereignisse können eingefügt werden und es ist möglich, vorhandene Zeilen zu aktualisieren oder zu löschen. Dabei gibt der Schlüssel eines Ereignisses an, welche Zeile geändert wird. Wie Streams sind auch Tabellen beständig, dauerhaft und fehlertolerant. Eine Tabelle verhält sich ähnlich wie eine materialisierte Ansicht, da diese automatisch geändert wird, sobald sich die Daten oder Tabellen ändern, anstatt dass direkte Einfüge-, Aktualisierungs- oder Löschvorgänge ausgeführt werden.

Tabellen können als Aktualisierungen in einer Datenbank betrachtet werden. Hier wird nur der aktuelle Zustand für jeden Schlüssel gespeichert.

Die Tabelle wird von Kafka Streams mithilfe eines Key-Value-Store materialisiert, der standardmäßig mit RocksDB implementiert wird. Durch das Konsumieren eines geordneten Streams von Ereignissen wird nur deren letztes Ereignis für jeden Schlüssel im clientseitigen Key-Value-Store aufbewahrt. Dadurch erhält man eine tabellenähnliche Darstellung der Daten.[35]

KTable

Eine KTable ist eine Abstraktion eines Changelog-Streams, bei dem jedes Ereignis eine Aktualisierung darstellt. Nur die letzte Darstellung eines bestimmten Schlüssels wird von der Anwendung getrackt. Da KTables partitioniert sind, enthält jede Kafka-Streams Aufgabe nur eine Teilmenge der vollständigen Tabelle.[35]

3.5.4 KGroupedTable

KGroupedTable ist eine Abstraktion einer gruppierten Tabelle. Es handelt sich um eine Zwischendarstellung nach einer Umgruppierung der Abstraktion KTable, bevor eine Aggregation auf die neuen Partitionen angewendet werden kann.[35]

GlobalKTable

Eine GlobalKTable ist eine Abstraktion eines Changelog-Streams, ähnlich wie die Abstraktion KTable.[35] Im Gegensatz zu einer KTable, die über alle Kafka Streams-Instanzen partitioniert ist, wird eine GlobalKTable vollständig pro Kafka Streams-Instanz repliziert. Jede Partition des zugrundeliegenden Themas wird von jeder GlobalKTable konsumiert, sodass eine vollständige Kopie der zugrunde liegenden Daten in jeder KafkaStreams-Instanz verfügbar ist.[50] Dies ermöglicht die Durchführung von joins mit einem KStream, ohne dass der Stream neu partitioniert werden muss.[35]

3.6 Apache Beam

Apache Beam geht auf das ursprüngliche MapReduce-System zurück, das Google 2004 in Form eines Papers veröffentlichte und die Art und Weise, wie verteilte Datenverarbeitung genutzt wird, grundlegend verändert hat. 2014 launchte Google das Projekt Google Cloud Dataflow, das auf Technologien beruhte, welche sich aus dem MapReduce-System entwickelt hatten. Es beinhaltete allerdings neuere Ideen wie eine erhöhte Abstraktion und einen Fokus auf Streaming sowie die Echtzeitausführung. 2016 wurde das Projekt von Google und einige Partner der Apache Software Foundation mit dem Namen Apache Beam übergeben.[23]

Apache Beam wird als ein einheitliches Programmiermodell bezeichnet, mit welchem es möglich ist, Batch- und Streaming-Pipelines zu implementieren. Das Projekt ist lizenziert unter der Open-Source-Lizenz.[26] Mit Beam ist es möglich, eine Event-Streaming-Verarbeitungs-Anwendung zu implementieren und diese mit verschiedenen Runnern in unterschiedlichen Event-Streaming-Verarbeitungssystemen auszuführen. Der Beam Runner ist dafür verantwortlich, die Datenverarbeitungspipeline zu übersetzen, sodass der Code mit dem unterliegenden Verarbeitungssystem kompatibel ist und ausgeführt werden kann.

Anstatt eine Anwendung für ein einzelnes System zu entwickeln, ermöglicht Apache Beam das Schreiben von Programmen, die mit jeder unterstützten Ausführungsengine kompatibel ist. Engine-spezifische Runner übersetzen den Apache Beam-Code in die Ziel-Laufzeitumgebung. Mit einer solchen Abstraktionsschicht ist theoretisch z.B. ein beliebiger Austausch von Engines möglich, ohne dass der Code angepasst werden muss.[31]

Dies lässt sich mit der Ausführung von C-Programmen auf unterschiedlichen Plattformen vergleichen, welche nur die C Standart Bibliotheken verwenden. Windows-APIs gibt es nicht in Linux und Linux-APIs gibt es nicht in Windows, daher muss der C Code vom Plattform spezifischen Kompiler kompiliert werden. Durch die Verwendung der Standard Bibliotheken kann die Anwendung kompiliert werden und auf einer Vielzahl von Plattformen ohne Modifikation ausgeführt werden.[56]

Die Beam-SDKs verwenden dieselben Klassen, um sowohl begrenzte als auch unbegrenzte Daten darzustellen und dieselben Transformationen, um mit diesen Daten zu arbeiten.[10]

Beam ist besonders nützlich für parallele Datenverarbeitungsaufgaben, bei denen das Problem in viele kleinere Datenpakete zerlegt werden kann, die unabhängig voneinander und parallel verarbeitet werden können. Apache Beam wird auch für Extrahier-, Transformier- und Ladeaufgaben (ETL) und reine Datenintegration verwendet. Diese Aufgaben sind nützlich, um Daten zwischen verschiedenen Speichermedien und Datenquellen zu verschieben, Daten in ein gewünschtes Format zu transformieren oder Daten in ein neues System zu laden.[10]

Eine positive Entwicklung in Apache Beam ist die Unterstützung mehrerer Programmiersprachen. Es ist möglich, Beam mit Java, Python, Go, Scala oder SQL zu entwickeln. Im Wesentlichen können Entwickler die Anwendungen in einer Programmiersprache ihrer Wahl schreiben.[26]

3.6.1 Abstraktionen

Das Beam-Modell besteht aus mehreren Kernabstraktionen, welche benutzt werden, um die Verarbeitungspipelines zu implementieren. Die wichtigsten Abstraktionen sind wie folgt:

- Pipeline

- PCollection
- PTransform

[23]

Eine Beam-Pipeline ist eine vollständige Beschreibung der Anwendung, welche ausgeführt werden soll. Die Pipeline enthält eine Beschreibung der Quellen, Operatoren, Transformationen und Senken und wie diese miteinander verbunden sind. Die Pipeline ist das Objekt, welches am Ende vom Beam Runner ausgeführt wird.[23]

Eine PCollection ist ein verteilter, unbeschränkter oder beschränker, unveränderlicher und homogener Datensatz.[53]

Ein PTransform stellt eine Datenverarbeitungsoperation in der Pipeline dar. PTranform ist eine Abstraktion für die Datentransformation. Es empfängt ein oder mehrere PCollection-Objekte und wendet die Transformationen auf diesen Daten an. Dies führt zu einer Erzeugung von mehreren oder gar keinen PColleciton-Objekten.[53]

Apache Beam bietet einige Kern Transformationen an, die abhängig vom Runner vollumfänglich oder nur teilweise in Apache Beam implementiert sind. Wie zum Beispiel:

- ParDo ist eine Element-für-Element-Verarbeitung von Daten. Dabei kann die Verarbeitung eines einzelnen Elements zu null oder mehreren Ausgabeelementen führen. Zusätzlich zu Standardoperationen wie Map oder Flat Map unterstützt ein ParDo auch Aspekte wie zustandsabhängige Verarbeitung.[10]
- GroupByKey, wie der Name schon sagt, werden Schlüssel-Werte-Paare verarbeitet. Alle Werte werden gesammelt, die zu demselben Schlüssel gehören. Für die Verwendung im Streaming-Mode muss ein Aggregations-Trigger oder ein nicht-globales Window verwendet werden, damit die Gruppierung auf einen endlichen Datensatz angewendet werden kann.[10]
- Flatten verschmilzt die Daten mehrerer PCollection Objekten zu einer einzigen PCollection[10]

Es ist wichtig zu beachten, dass Beam nicht zwischen kontinuierlichen und unendlichen Daten und der Batchverarbeitung unterscheidet. Die Batchverarbeitung wird als ein Spezialfall der Stream Verarbeitung angesehen. Für beide werden die gleichen Primitiven verwendet, zudem können die Runner das Wissen nutzen, ob bestimmte Quellen begrenzt oder unbegrenzt sind, um die Ausführung des Modells zu beschleunigen.[53]

3.6.2 Runner

Ein Runner ist die Software, die eine Pipeline annimmt und ausführt. Die meisten Runner sind Übersetzer oder Adapter für parallele Big-Data-Verarbeitungssysteme.[52] Ein Runner ist dafür verantwortlich, Beam-Pipelines so zu übersetzen, dass sie auf einer Ausführungsmaschine ausgeführt werden können.[26] Der Runner erhält den Graphen und übersetzt die PTranform Funktion in den nativen Code des Event-Streaming-Verarbeitungssystem, sodass dieser dort ausgeführt werden kann. Eine ParDo Funktion in Apache Beam kann z.B. in eine FlatMap Funktion in Apache Spark übersetzt werden. Siehe: 3.3

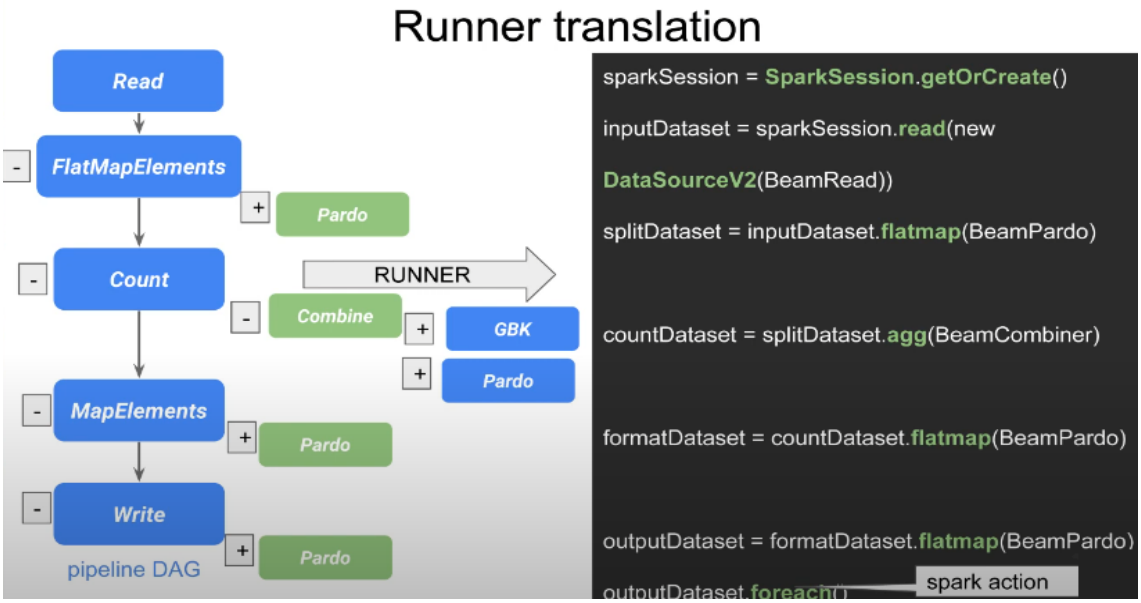

Abbildung 3.3: Übersetzung Runner -> System[17]

Die Idee ist, dass die Pipeline nur einmal geschrieben werden muss und diese entweder mit Batch- oder Streaming-Daten in verschiedenen Stream-Verarbeitungssystemen ausgeführt werden kann. Wenn diese ausgeführt werden soll, wird eines der unterstützten Systeme zur Verarbeitung der Daten ausgewählt. Eine große Integrationstestsuite in Beam namens 'ValidatesRunner' stellt sicher, dass die Ergebnisse gleich sind, unabhängig davon, welches Backend für die Ausführung gewählt worden ist. [26]

Beam unterstützt derzeit(Februar 23) die folgenden Runner:

- Direct Runner

- Apache Flink
- Apache Nemo
- Apache-Samza
- Apache Spark
- Google Cloud Dataflow
- Hazelcast-Jet
- Twister2

[10]

Diese Gruppe umfasst beides Closed-Source-Systeme wie Google Cloud Dataflow und IBM Streams als auch Open-Source-Systeme wie Apache Flink und Apache Spark. Daher ist Apache Beam ein weit verbreitetes Projekt mit einer hohen Relevanz.

Fähigkeitsmatrix der verschiedenen Runner

Es gibt jedoch gewisse Einschränkungen bei den Funktionalitäten der einzelnen Runner. Apache Beam bietet eine portable API-Schicht für den Aufbau von parallelen Verarbeitungspipelines, die auf einer Vielzahl von Stream-Verarbeitungssystemen ausgeführt werden können. Die Kernkonzepte dieser Schicht basieren auf dem Beam-Modell und sind in jedem Beam-Runner in abweichendem Maße implementiert. Eine Fähigkeitsmatrix wurde von Apache Beam erstellt, um die Fähigkeiten der einzelnen Runner zu verdeutlichen.[9]

Viele Funktionalitäten sind nur teilweise implementiert und es kann vorkommen, dass die Pipeline mit einem bestimmten Runner nicht ausgeführt werden kann. Bevor man etwas mit einem Runner ausführen möchte, sollte man sich die Fähigkeitsmatrix von Apache Beam angucken. Zustandsbehafteten Verarbeitungen sind in Beam z.B. nur mit dem Spark, Flink, Samza, Hazelcast-Jet und dem Google Cloud Dataflow Runner möglich.

In der Fähigkeitsmatrix[9] ist gut zu erkennen, dass der Google Cloud Dataflow Runner und der Apache Flink Runner am umfänglichsten implementiert sind. Bei diesen beiden Runnern sind nahezu alle Funktionalitäten implementiert, nur wenige Funktionalitäten sind nur zum Teil umgesetzt.

Der 'Direct Runner' führt Pipelines auf dem lokalen Rechner aus und wurde entwickelt, um zu validieren, dass Pipelines dem Apache Beam-Modell so genau wie möglich entsprechen. Anstatt sich auf die effiziente Ausführung der Pipeline zu konzentrieren, führt der Direct Runner zusätzliche Überprüfungen durch, um sicherzustellen, dass sich die Anwender auf Semantiken und Garantien verlassen, die durch das Modell garantiert werden.[11]

Apache Beam empfiehlt, die Pipelines mit dem DirectRunner zu testen, um sicherzustellen, dass die erzeugten Pipelines über verschiedene Runner hinweg robust sind.[11]

Einige dieser Prüfungen beinhalten:

- Durchsetzung der Unveränderlichkeit von Elementen
- Durchsetzung der Kodierbarkeit von Elementen
- Elemente werden an allen Stellen in beliebiger Reihenfolge verarbeitet
- Serialisierung von Benutzerfunktionen (DoFn, CombineFn, etc.)

[11]

Die Verwendung des Direct Runner zum Testen und Entwickeln trägt dazu bei, dass die Pipelines über verschiedene Beam Runner hinweg stabil sind. Darüber hinaus kann das Debuggen von fehlgeschlagenen Ausführungen eine nicht triviale Aufgabe sein, wenn eine Pipeline z.B. auf einem Remote-Cluster ausgeführt wird. Es ist oft schneller und einfacher, lokale Unit-Tests auf dem implementierten Pipeline-Code auszuführen. Wenn die Pipeline lokal getestet wird, können auch bevorzugte lokalen Debugging-Tools verwendet werden.[11]

3.6.3 Apache Flink

Der Apache Flink Runner kann verwendet werden, um Beam-Pipelines mit Apache Flink auszuführen. Der Flink Runner und Flink eignen sich für große, kontinuierliche Datenverarbeitungen und bietet Folgendes:

- Eine System, welches die Batchverarbeitung als auch die Verarbeitung von kontinuierlichen Daten unterstützt

- Eine Laufzeit, die gleichzeitig einen sehr hohen Durchsatz und eine geringe Ereignislatenz unterstützt
- Eine Fehlertoleranz mit einer einmaligen Verarbeitungsgarantie (exactly-once)
- Backpressure wird unterstützt. Darunter zählt z.B. das Verwerfen oder Buffern von Ereignissen
- Benutzerdefinierte Speicherverwaltung für effizientes und robustes Umschalten zwischen In-Memory- und External-Memory(out-of-core) Datenverarbeitungsalgorithmen

[11]

Den Flink-Runner gibt es in zwei Varianten:

- Den ursprünglichen klassischen Runner, der nur Java (und andere JVM-basierte Sprachen) unterstützt
- Den portable Runner, welcher Java,Python und Go zur Verfügung stellt

[11]

Beam und seine Runner unterstützten ursprünglich nur JVM-basierte Sprachen (z.B. Java/Scala/Kotlin). Die Python- und Go-SDKs wurden erst später hinzugefügt. Die Runner und deren Architektur musste erheblich geändert werden, um die Ausführung von Pipelines in anderen Sprachen zu unterstützen.[11]

Bei Anwendungen, die nur Java verwenden, sollte der klassische Runner verwendet werden. Der portable Runner wird den klassischen Runner in Zukunft ersetzen, da dieser das standardisierte Framework für die Ausführung von Java, Python und Go enthält.[11]

Es gibt einige Gründe, warum man Apache Beam mit Apache Flink verwendet, anstatt nur Flink zu benutzen. Letztlich ergänzen sich Beam und Flink gegenseitig und bieten dem Nutzer einen zusätzlichen Nutzen. Die Hauptgründe für die Verwendung von Beam mit Flink sind die folgenden:

- Beam bietet eine einheitliche API sowohl für die Batch- als auch für die Streamverarbeitung.
- Beam kommt mit nativer Unterstützung für verschiedene Programmiersprachen wie Python oder Go und mit all ihren Bibliotheken z.B. Numpy oder Pandas

- Die Leistung von Apache Flink, wie z.B. die exactly-once Semantik, die starke Speicherverwaltung und die Robustheit
- Beam-Programme laufen auf der bestehenden Flink-Infrastruktur oder der Infrastruktur anderer unterstützter Runner wie Spark oder Google Cloud Dataflow.
- Zusätzliche Funktionen wie Side-Inputs und sprachübergreifende Pipelines, die nicht nativ in Flink, sondern nur bei der Verwendung von Beam mit Flink unterstützt werden

[26]

Wie wird Flink in Beam ausgeführt?

Der Flink-Runner wird auf unterschiedliche Weise übersetzt. Je nachdem, ob die Ausführung im Batch- oder Streaming-Modus erfolgt, übersetzt der Runner entweder in die DataSet- oder in die DataStream-API von Flink. Aufgrund der Ünterstützung der Mehrsprachigkeit wurden zwei weitere Übersetzungspfade hinzugefügt:

- Der klassische Flink-Runner für Batch-Jobs: Führt Batch-Java-Pipelines aus
- Der klassische Flink-Runner für Streaming-Jobs: Führt Streaming-Java-Pipelines aus
- Der Portable Flink Runner für Batch-Jobs: Führt sowohl Java als auch Python, Go und andere unterstützte SDK-Pipelines für Batch-Szenarien aus
- Der portable Flink-Runner für Streaming-Jobs: Führt sowohl Java als auch Python, Go und andere unterstützte SDK-Pipelines für Streaming-Szenarien aus

[26]

Der klassische Flink-Runner war die erste Version des Runners. Beam-Pipelines werden als Graphen in Java dargestellt, welche sich aus den zusammengesetzten und primitiven Transformationen ergeben. Der Graph wird mithilfe von Beam in eine topologische Ordnung gebracht.[26] Ein Graph mit einer topologischen Ordnung wird als eine Reihenfolge von Dingen bezeichnet, bei dem vorgegebene Abhängigkeiten erfüllt sind. Anschließend wird der Graph transformiert und der Flink Runner generiert die API-Aufrufe, welche

normalerweise beim Schreiben eines Flink-Jobs geschehen.[26] Trotz der Gemeinsamkeiten und sehr ähnlichen Konzepte von Apache Beam und Apache Flink gibt es viele Unterschiede. Diese machen es unmöglich, Beam-Pipelines 1:1 in ein Flink-Programm zu übersetzen.[26] Ein Beispiel dafür ist die Serialisierung. Wenn Daten in Flink übertragen werden, müssen diese zunächst in Bytes umgewandelt werden. Dies geschieht mithilfe von Serializern. Flink hat ein System, um den richtigen Codierer für einen bestimmten Typ zu instanziieren z.B. IntTypeSerializer für einen Integer. Apache Beam hat auch ein eigenes System, welches dem von Flink ähnelt, aber eine etwas andere Schnittstellen verwendet. Serialisierer werden in Beam Coder genannt. Um einen Beam Coder in Flink auszuführen, muss dieser kompatibel gemacht werden.[26]

Der portable Flink-Runner in Beam, ist die Weiterentwicklung des klassischen Runners. In der Abbildung: 3.4 ist links der klassischen Runner visualisiert und daneben der Portable Runner. Klassische Runner sind an das JVM-Ökosystem gebunden, der portable Runner von Beam überwindet dies und ermöglicht die Entwicklung in Python, Go und andere Programmiersprachen. Durch die Möglichkeit, Beam in mehreren Programmiersprachen zu entwickeln, musste die Übersetzungslogik des Runners geändert werden. [26]

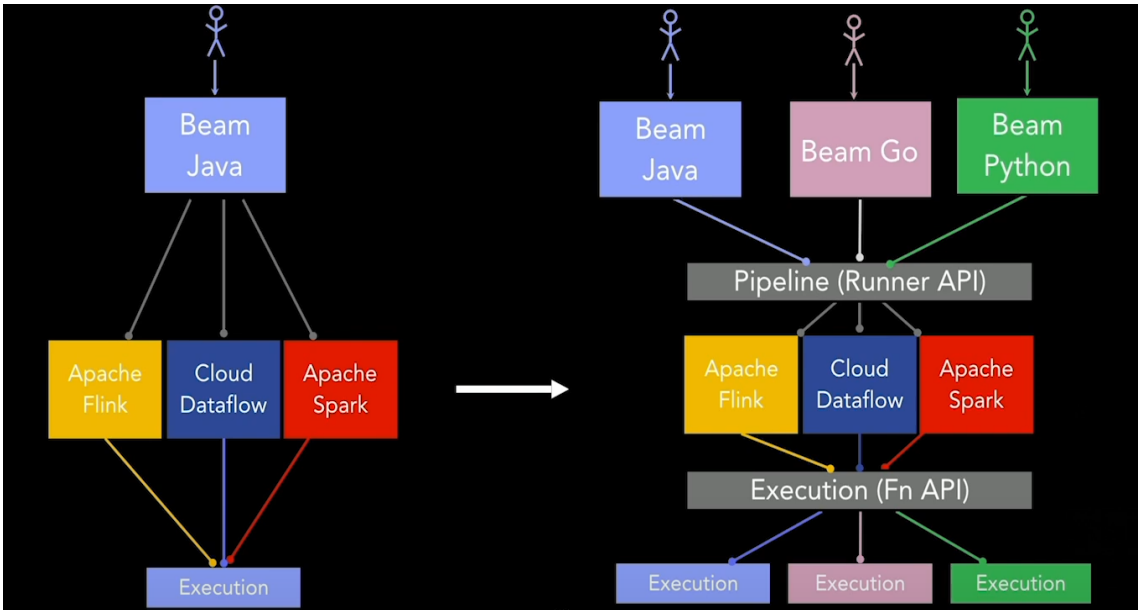

Abbildung 3.4: Beam Sprachen Portabilität[26]

Es gibt zwei wichtige Bausteine für den portierbaren Runner: Siehe: 3.4

- Ein gemeinsames Pipeline-Format für alle Sprachen: Die Runner-API
- Eine gemeinsame Schnittstelle für die Kommunikation zwischen dem Runner und dem in einer beliebigen Sprache geschriebenen Code: Die Fn-API

[26]

Die Runner-API bietet eine universelle Darstellung der Pipeline als Protobuf, die die Transformationen, Typen und den Anwendercode enthält. Protocol Buffers (Protobuf) ist ein von Google entwickeltes Format zur Serialisierung von Daten.[28] Protobuf wurde als Format gewählt, weil jede Sprache über Bibliotheken für dieses Format verfügt.[26] Für den Ausführungsteil hat Beam die Fn-API-Schnittstelle eingeführt, um die Kommunikation zwischen dem Runner und dem Benutzercode zu ermöglichen, der in einer anderen Sprache geschrieben sein kann und in einem anderen Prozess ausgeführt wird.[26] Die Fn-API ist hochgradig abstrakt und umfasst mehrere generische Komponenten wie z.B. den 'Data service' oder den 'State service'.

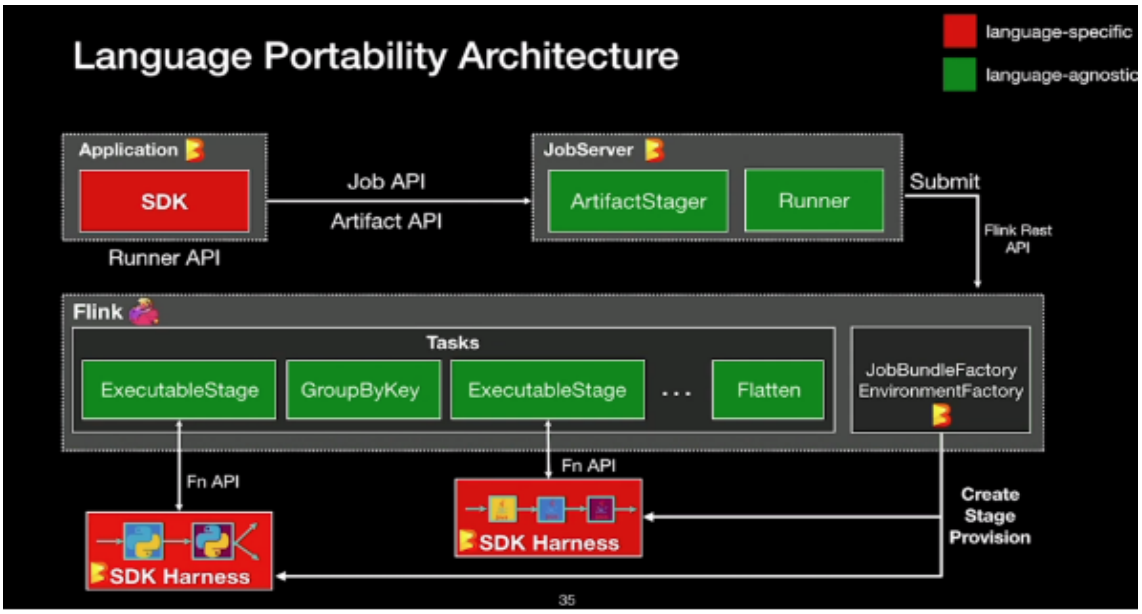

Abbildung 3.5: Runners- and FN-API[26]

Wenn das Python SDK verwendet wird, um eine Pipeline zu schreiben, wird diese mithilfe der Runner-API in eine einheitliche Pipeline übersetzt. Anschließend wird die übersetzte Anwendung über die Job-API an den JobServer von Beam übermittelt. Der JobServer ist ebenfalls eine Beam-Komponente, die sich um die Bereitstellung der erforderlichen

Abhängigkeiten während der Ausführung kümmert. In der Pipeline existieren Java Abhängigkeiten, welche von Beam genutzt werden, und Python Abhängigkeiten. Der Job-Server hat einen 'ArtifactStager', welcher die Python Pipeline erhält, die Abhängigkeiten ermittelt und diese in z.B. Datei-Systemen bereitstellt.[26]

Die Übersetzung wird vom Job-Server mit der sogenannte ExecutableStage-Transformation angestoßen. Siehe: 3.5 Dabei handelt es sich im Wesentlichen um eine ParDo-Transformation, die für die Aufnahme von sprachabhängigem Code konzipiert ist. Beam versucht, möglichst viele dieser Transformationen in einer 'ExecutableStage' zu kombinieren. Das Ergebnis ist ein Flink-Programm, das an den Flink-Cluster gesendet und dort ausgeführt werden kann.[26] Der Hauptunterschied zum klassischen Runner besteht darin, dass während der Ausführung eine Umgebungen gestartet wird, um die genannten ExecutableStages auszuführen.[26] ExecutableStages beeinhalten z.B. Python spezifische Funktionen oder Abhängigkeiten. Eine Docker-basierte Umgebung ist z.B. eine verfügbare Beam Umgebung, um ExecutableStages auszuführen.[26] Die Umgebungen enthalten das SDK Harness, d.h. den Code, der die Ausführung und die Kommunikation mit dem Runner über die Fn-API übernimmt.[26] Durch die erhöhte Kommunikation und Übersetzung ist die Pipeline in Java performanter als dieselbe Pipeline, welche in Python implementiert worden ist.[26]

3.6.4 Apache Spark

Der Apache Spark Runner kann verwendet werden, um Beam-Pipelines mit Apache Spark auszuführen. Der Spark Runner kann Spark-Pipelines genau wie eine native Spark-Anwendung ausführen.

Der Spark Runner bietet:

- Batch- und Streaming-Pipelines
- Die gleichen Fehlertoleranzgarantien wie bei RDDs und DStreams.
- Integriertes Metrik-Reporting mit dem Metrik-System von Spark, das auch Beam-Aggregatoren auswertet
- Native Unterstützung für Beam-Side-Inputs über die Broadcast-Variablen von Spark

[11]

Der Spark Runner wird in verschiedenen Varianten angeboten. Beim Apache Spark Runner kann man auswählen, welche API benutzt werden soll, die Spark Streaming API oder die Structured Streaming API.[11] Bei der Structured Streaming API wird jedoch kein 'Streaming-Mode' unterstützt, sondern nur die Verarbeitung im 'Batch-Mode'. Zudem sind die Verarbeitungen mit einem Zustand für den Spark Streaming Runner nur teilweise implementiert und verfügbar. Die Funktion DoFn mit einem Zustand ist zum Beispiel noch nicht mit dem Spark Streaming Runner verfügbar. Um mit dem Spark Runner und Zuständen zu arbeiten, muss eine CombineFn Funktion implementiert werden, welche eine Sammlung von Eingabewerten zu einem einzigen Ausgabewert kombiniert.

4 Experiment

Es wurde ein Experiment in den vorgestellten Event-Streaming-Verarbeitungssystemen implementiert, um verschiedene Metriken zu erhalten und diese auszuwerten.

Um die verschiedenen Abstraktionen und Systeme zu testen, wurde ein Experiment durchgeführt. Dabei stand die Messung der Zeiten, die Verarbeitungsdauer, das Lesen und Schreiben der Daten in den jeweiligen Systemen und den darunterliegenden Abstraktionen im Fokus.

Es gibt zwei verschieden Haupttypen von Even-Streaming-Verarbeitungssystemen: Deklarative Systeme und kompositionelle Systeme.

In deklarativen Systemen wie Apache Spark und Flink ist der Anwendungscode sehr funktional. Der Benutzer impliziert durch den Anwendungscode einen gerichteten azyklischen Graph (DAG), welcher von der Engine optimiert werden kann. In Apache Spark SQL wird hierfür z.B. der Catalyst-Optimierer benutzt.

Bei kompositionellen Systemen wie Apache Storm befindet sich der Anwendungscode auf einer niedrigeren Ebene, da der Benutzer den DAG explizit definiert. Daraus resultiert oft ein leicht ineffizienter Code, jedoch unterliegt der Code der vollständigen Kontrolle des Entwicklers. In Apache Storm resultiert der DAG durch die Implementation von Spouts, Bolts und der Topologie.

Das Experiment umfasst deklarative sowie kompositionelle Systeme.

4.1 Forschungsfragen

Durch das Experiment sollen verschiedenen Forschungsfragen bezüglich Event-Streaming-Verarbeitungssysteme beantwortet werden.

Komplexität

Wie komplex und aufwändig ist es, einen Event-Stream-Verarbeitungsjob in den verschiedenen Event-Streaming-Verarbeitungssystemen zu implementieren.
Forschungsfrage 1:
Erleichtert eine hohe Abstraktionsschicht die Implementierung der Streaming-Jobs in den Verarbeitungssystemen?

Latenz

Wie schnell erhält man Antworten und wie lange dauert es, dass ein Ereignis das Event-Streaming-Verarbeitungssystem durchläuft.
Forschungsfrage 2:
Wie schnell ist die Verarbeitung und die Ereignislatenz der verschiedenen Ereignisse innerhalb der Event-Streaming-Verarbeitungssysteme?

Apache Beam als Abstraktionsschicht

Vergleich der Latenz zwischen einem nativen Event-Streaming-Verarbeitungssystem wie Apache Flink und der Ausführungsengine des Flink Runners von Apache Beam.
Forschungsfrage 3:
Wie wirkt sich die Nutzung von Apache Beam auf die Latenzzeiten und die Komplexität aus, steigen die Latenzzeiten und sinkt die Komplexität durch die Abstraktionsschicht?

4.2 Komplexität

Der Begriff Komplexität wird in vielen Bereichen verwendet, in der Regel wird dies zur Beschreibung von Systemen benutzt, welche die menschliche Verständlichkeit beeinflussen.

Code Komplexität wird von Basili hingegen als ein Maß für die Ressourcen definiert, die ein System oder ein Mensch bei der Interaktion mit einer Software zur Erfüllung einer bestimmten Aufgabe benötigt.[6]

Um die Komplexität von dem Stream-Verarbeitungsjob und den darunterliegenden Systemen zu bewerten, soll zu einem die Metrik, wie viele Programmierzeilen(Lines of code) benötigt werden, herangezogen werden.[6] Des Weiteren sollen die Informationen, welche es im Internet/Büchern zu den Systemen zu finden gibt, also die Größe der 'Community' und den Aufwand, um den Job zu implementieren, in die Bewertung mit einfließen.

4.3 Latenz

Es gibt viele verschiedene Ansätze, wie die Latenz in einem System gemessen werden kann. Viele Wissenschaftler haben sich mit diesem Thema beschäftigt und unterschiedliche Frameworks entwickelt, um die Performance eines verteilten Event-Streaming-Verarbeitungssystems zu testen.

Ein Framework zum Testen von verteilten Event-Streaming-Verarbeitungssystemen ist DSPBench. DSPBench ist eine Benchmark-Suite mit 15 Anwendungen aus verschiedenen Bereichen. Es ist eine reichhaltige Suite von Anwendungen, die der Community in einem GitHub-Repository öffentlich zur Verfügung gestellt wurde. Es wurden Anwendungen ausgewählt, die ein breites Spektrum abdecken, darunter Finanzen, Netzwerküberwachung, Verkehrsüberwachung, Werbung, Spiele, Telecom, soziale Netzwerke sowie Sensoren. Zudem ist DSPBench eine Low-Level-API für die einheitliche Entwicklung von Anwendungen, die es ermöglicht, diese einmal zu schreiben und überall auszuführen, solange die spezifischen Adapterkomponenten ebenfalls entwickelt wurden.[13]

Eines der Hauptziele der Suite ist es, den Vergleich zwischen verschiedenen verteilten Event-Streaming-Verarbeitungssystemen zu ermöglichen. Zwei Komponenten der Architektur sind die Input- und Output-Komponenten. Die erste ist für die Versorgung der Anwendungen mit kontinuierlichen Daten zuständig, während die zweite die Ergebnisse speichert, damit diese analysiert werden können. Die Input-Komponente wird mit Apache Kafka umgesetzt.

DSPBench misst die für die Verarbeitung eines Tupels pro Operator, die benötigte Zeit, die Größe der Tupel bei jedem Operator und die Speichernutzung der Anwendungen.[13]

Latenz bezeichnet die Verzögerung, die das verteilte Event-Streaming-Verarbeitungssystem vom Eintreffen des Ereignisses bis zur Erzeugung des Ergebnisses verursacht. Unabhängig davon, wie schnell das Event-Streaming-Verarbeitungssystem Ereignisse verarbeitet, kann ein Operator ein Ereignis erst dann erzeugen, wenn alle beitragenden Ereignisse

von der Quelle außerhalb des Systems eingetroffen sind. Die Latenzzeit ist intuitiv das Zeitintervall von diesem Zeitpunkt bis zu dem Zeitpunkt, an dem das Ereignis tatsächlich vom Operator produziert wird.[16]

Alle Tupel, die in den Graphen fließen, erben in DSPBench den Erstellungszeitstempel des vorgelagerten Tupels. Die Latenzzeit wird in DSPBench als die Differenz zwischen der Zeit, wann das Tupel geschrieben worden ist und dem Erstellungszeitstempel, der im empfangenen Tupel enthalten ist, gemessen. Es ist wichtig, dass die Systeme eine kohärente Zeit im Cluster aufweisen. Dies wird durch das Network Time Protokoll (NTP) gewährleistet.[13]

2017 wurde in einem Paper ein weiteres Framework vorgestellt.[38] Bei diesem Framework soll ein System, welches getestet werden soll, unabhängig vom Message-Broker getestet werden. Zudem wurden hier verschiedene Metriken bezüglich verschiedenen Latenzen in verteilten Event-Streaming-Verarbeitungssystemen definiert.

Moderne Event-Streaming-Verarbeitungssysteme unterscheiden zwei Begriffe von Zeit: Event-Time und Processing-Time. Die Event-Time ist der Zeitpunkt, an dem ein Ereignis erfasst wird, während die Processing-Time der Zeitpunkt ist, an dem ein Operator ein Tupel verarbeitet hat.[38]

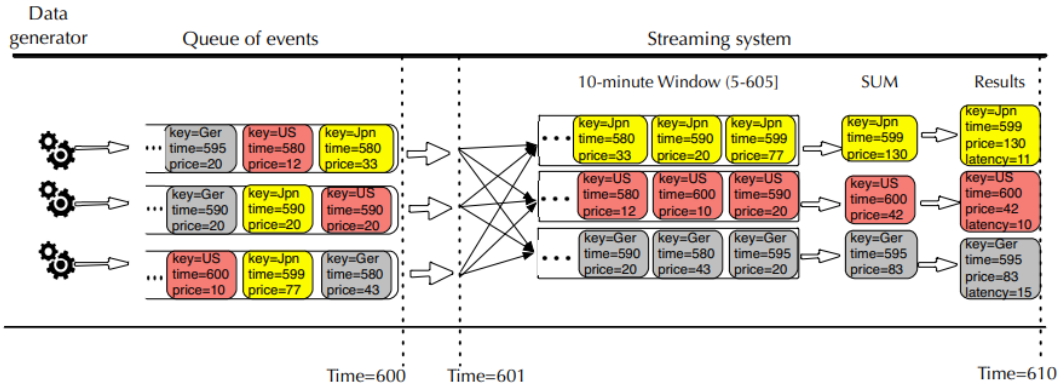

Abbildung 4.1: Latenz[38]

Die Event-Time-Latenz wird als das Intervall zwischen der Event-Time eines Tupels und seiner Emissionszeit vom ‘System-under-test’(SUT)-Ausgabeoperator definiert.[38] Proccesing-Time-Latenz wird als die Zeitspanne zwischen dem Zeitpunkt, an dem das Ereignis den Eingangsoperator des Streaming-Systems erreicht hat und dem Ausgabezeitpunkt vom Ausgangsoperator des Systems, welches getestet wird, bezeichnet.[38]

Die Event-Time-Latenz umfasst die Zeit, die ein bestimmtes Ereignis in einer Warteschlange verbracht hat, um verarbeitet zu werden, während die Proccesing-Time-Latenz die Zeit misst, die für die Verarbeitung des Ereignisses durch das Event-Streaming-Verarbeitungssystem benötigt wird. In praktischen Szenarien ist die Event-Time-Latenz sehr wichtig, da sie die Zeit definiert, in der der Benutzer mit einem bestimmten System interagiert.[38] Die Event-Time von zustandsbehafteten Ereignissen ist die maximale Event-Time aller Ereignisse, die zu der Ausgabe beigetragen haben. Dies ist in der Abbildung: 4.2 dargestellt. Die Ereignisse werden mithilfe von einem Message-Broker in das System geladen. Vom Message-Broker erhalten die Ereignisse einen Zeitstempel. Dieser wird hier in den Ereignissen 'time' deklariert. Anschließend werden die Ereignisse im System geladen, dort erhalten sie einen weiteren Zeitstempel. Wenn die Verarbeitung abgeschlossen ist, wird die Processing-Time-Latenz berechnet, welche angibt, wie lange die Verarbeitung gedauert hat. Die Processing-Time von zustandsbehafteten Ereignissen ist die maximale Processing-Time aller Ereignisse, die zu dem Ergebnis beigetragen haben.

Ein weiteres Framework ist ESPBench, dies ist ein Leistungsbenchmark Framework mit umfassender Tool-Unterstützung. ESPBench deckt alle Kernfunktionalitäten von verteilten Event-Streaming-Verarbeitungssystemen ab, einschließlich der Kombination von Streaming-Daten mit strukturierten Geschäftsdaten. Bei ESPBench wird ein SUT ähnlich wie in[38] unabhängig vom Message-Broker und dem Benchmark Framework getestet. Dafür wird das System, welches getestet wird, von den restlichen Komponenten losgelöst und interagiert nur mit dem Message-Broker.

Das System liest Eingabedaten aus Apache Kafka und schreibt die Ergebnisse je nach Abfrage, entweder zu Apache Kafka oder an ein Datenbankmanagement System wie Postgres zurück. Innerhalb des ESPBench Frameworks existiert ein Validator und ein Ergebniskalkulator, diese bestimmen die Korrektheit der Abfrageantworten und berechnen die Benchmark-Ergebnisse. Das Tool ermittelt aggregierte Ergebnisse wie die mittlere Latenz sowie die einzelnen Latenzen für jedes Ergebnis. ESPBench schreibt die Ausgabe in Log- und CSV-Dateien, die für weitere Analysen verwendet werden können wie das Plotten einzelner Latenzen. Der Validator liest die Eingabedaten, errechnet die Abfrageergebnisse und vergleicht sie mit der Ausgabe des Systems. Außerdem berechnet das Framework Ergebnislatenzen, indem Zeitstempel-Differenzen vom Validator berechnet werden. Um die Latenzen zu berechnen, werden die Zeitstempel von den Ergebnissen sowie von Kafka oder dem DBMS benutzt. Dieses Konzept ermöglicht die erwähnten objektiven Leistungsmessungen außerhalb des SUT.

4.4 Vorgehensweise

Das Ziel ist es, die Abstraktionen und Komplexität in den Event-Streaming-Verarbeitungssystemen unabhängig vom Message-Broker zu testen. Alle Systeme erhalten Daten über Apacha Kafka Themen. Apache Kafka ist ein Publish-Subscribe Nachrichtendienst, mit dem es möglich ist, Daten in Themen zu schreiben und diese Daten in einem externen System zu erhalten. Apache Spark, Apache Storm, Apache Flink, KsqlDB und Kafka Streams haben eine Integration mit Kafka und es gibt mehrere Benchmarking Frameworks (siehe: 4.3), die Apache Kafka als Message-Broker benutzen.

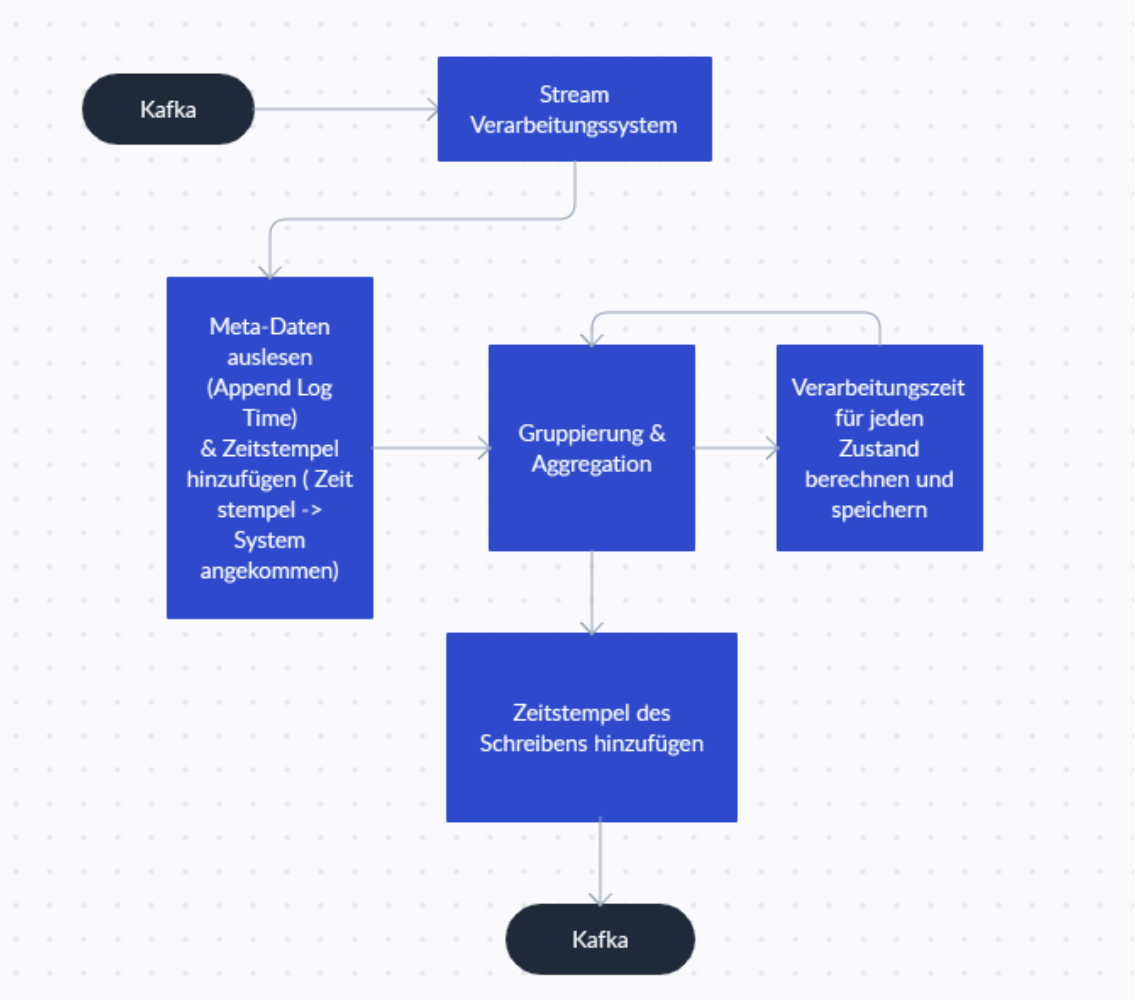

Abbildung 4.2: Ablauf Experiment

Beim Schreiben der Daten in Kafka wird jedem Ereignis ein Zeitstempel hinzugefügt. Zudem fügt Kafka jedem Ereignis ein Zeitstempel in den Metadaten hinzu, standardmäßig wird die Event-Time des produzierenden Systems benutzt. Es ist jedoch auch möglich, Kafka so zu konfigurieren, dass die 'AppendLogTime' als Zeitstempel benutzt wird. Dieser Zeitstempel gibt an, wann das Ereignis beim Kafka-Broker angekommen ist.(Siehe: 4.3) Der Kafka-Broker ist zuständig für die Partitionierung der Daten und für Lese- und Schreibanfragen der Daten.

In jedem System wird beim Erhalt der Ereignisse ein weiterer Zeitstempel hinzugefügt, dieser gibt an, wann das Ereignis im Event-Streaming-Verarbeitungssystem angekommen ist. (Siehe: 4.3) Anschließend werden die eintreffenden Daten verarbeitet. In jedem System werden dieselben Transformationen und Aktionen ausgeführt, sodass dieselben Ergebnisse bei denselben Daten zu erwarten sind.

Wenn der Streaming-Job zustandsbezogene Transformationen durchführt, wird der neueste Zeitstempel vom Kafka-Broker für jeden Zustand benutzt, um die Event-Time zu messen. Für die Verarbeitungszeit von zustandsbezogenen Transformationen wird für jeden Zustand die Zeit berechnet, die jedes Event benötigt hat, um verarbeitet zu werden. Als Verarbeitungszeit wird die maximale Verarbeitungszeit für jeden Zustand benutzt und diese wird in dem jeweiligen Event-Streaming-Verarbeitungssystem zu dem Ergebnis hinzugefügt.(Siehe: 4.3) Bevor die Daten in ein Kafka Thema geschrieben werden, wird ein weiterer Zeitstempel zu dem verarbeiteten Ereignis hinzugefügt. Dieser gibt an, wann das Ereignis vom Event-Streaming-Verarbeitungssystem in eine Datenbank oder ein Thema von Kafka geschrieben worden ist. (Siehe: 4.3)

Das Ergebnis ist mit verschiedenen Zeitstempeln und der Verarbeitungszeit angereichert. Daraufhin ist es möglich, die Event-Time-Latenz anhand der Differenz der verschiedenen Zeitstempel zu berechnen.

Die Latenzzeiten werden im nächsten Abschnitt definiert.

4.4.1 Processing-Time-Latenz

Es wird die Zeit gemessen, die ein Ereignis im Event-Streaming-Verarbeitungssystem benötigt, um verarbeitet zu werden. Dies wird berechnet, indem der Zeitpunkt, wann das Ereignis im System angekommen ist, von dem Zeitpunkt des fertig verarbeiteten Ereignisses abgezogen wird.

4.4.2 Event-Time-Latenz

Die Event-Time-Latenz wird berechnet, indem die 'Append Log Time' vom Kafka-Broker vom Zeitpunkt der Ausgabe des verarbeiteten Ereignisses subtrahiert wird.

4.4.3 Event-Time von zustandsbehafteten Ereignissen

Der neueste zugehörige Zeitstempel, die Append Log Time vom Kafka-Broker, wird für alle unterschiedlichen Zustände benutzt. Dieser wird vom Zeitpunkt der Ausgabe subtrahiert. Anschließend erhält man die Event-Time-Latenz von zustandsbehafteten Ereignissen.

4.4.4 Processing-Time von zustandsbehafteten Ereignissen

Bei jeder Verarbeitung eines Zustandes wird die Verarbeitungszeit gemessen. Dies geschieht, indem der Zeitstempel vom Eintreffen des Ereignisses im Event-Streaming-Verarbeitungssystem von dem Zeitstempel, wann es fertig verarbeitet wurde, abgezogen wird. Für jeden Zustand wird die maximale Verarbeitungszeit gespeichert und dem aggregierten Ergebnis angereichert. Anschließend erhält man die Proccesing-Time-Latenz von zustandsbehafteten Ereignissen.

4.5 Implementation

Das Experiment wurde in fünf verschiedenen Event-Streaming-Verarbeitungssystemen implementiert. Alle Systeme wurden mit Docker und in Java implementiert, um eine einheitliche Voraussetzung zu schaffen. Die Anwendungen laufen auf einer Maschine mit 24 GB DDR3 Arbeitsspeicher und einem Intel Core i5-8600K Prozessor mit 6 Kernen. Es wurden keine lokalen Ressourcengrenzen für Docker definiert.

Als Erstes werden die Daten mithilfe von einem Pythonskript ausgelesen und in ein Kafka-Thema geschrieben. Das Pythonskript schreibt die Daten ohne Drosselung in ein definiertes Kafka-Thema.

Der Streaming-Job im jeweiligen Event-Streaming-Verarbeitungssystem lauscht auf Nachrichten des Kafka-Themas und fügt beim Eintreffen des Ereignisses einen Zeitstempel

hinzu und liest die Meta-Daten des Ereignisses, um die Append Log Time vom Kafka-Broker zu erhalten. Anschließend werden die Daten verarbeitet, die Vorgehensweise ist in fast allen ausgewählten Systemen dieselbe. Lediglich in Apache Spark und KSQLDB ist die Herangehensweise anders.

Anschließend werden die verarbeiteten Daten vom Event-Streaming-Verarbeitungssystem in ein Kafka-Thema zur Auswertung geschrieben. Alle Streaming-Jobs wurden 10x ausgeführt und die Daten wurden mithilfe von einem Pythonskript ausgewertet, analysiert und visualisiert.

Versionen der Event-Streaming-Verarbeitungssysteme

Die Event-Streaming-Verarbeitungssysteme wurden mit den folgenden Versionen implementiert und ausgewertet.

System	Version
Apache Kafka	3.3.0
Apache Storm	2.4.0
Apache Spark	3.2.0
Apache Flink	1.15.2
KsqlDB-Server	0.28.2
Apache Beam	2.45.0

4.5.1 Daten

Der Datensatz enthält eine Liste von Videospielen, welche in den Jahren von 1980 bis 2020 mehr als 100.000 Exemplaren verkauft haben.

Der Datensatz beinhaltet 16179 Ereignisse und jedes Ereignis hat die folgenden Attribute:

- Name - Der Name des Spiels
- Platform - Die Platform (PC,Playstation)
- Genre - Genre des Spiels
- Jahr - Erscheinungsdatum des Spiels

- Publisher - Herausgeber
- NA Sales - Verkäufe in Nord Amerika
- EU Sales - Verkäufe in Europa
- Other Sales - Verkäufe im Rest der Welt
- Global Sales - Insgesamt global verkauft

Die Anzahl der Spiele pro Jahr unterscheiden sich enorm. Im Jahr 2020 gibt es z.B. nur ein Spiel, welches im Datensatz vorhanden ist. Im Jahr 1988 gibt es hingegen 1427 Spiele. Da die Daten nach ihrem Jahr gruppiert werden, unterscheidet sich die Anzahl der Ereignisse, die für das jeweilige Jahr vom System verarbeitet werden müssen.

Zustand	Anzahl Ereignisse
2020	1
2017	3
1980	9
1984	14
1985	14
1990	15
1987	16
1989	17
1983	17
1986	21
1982	36
1991	41
1992	43
1981	46
1993	62
1994	121
1995	219
1996	263
N/A	269
1997	289
1999	338
2000	350
1998	379
2001	482
2016	502
2013	544
2014	581
2015	606
2012	653
2004	762
2003	775
2002	829
2005	939
2006	1006
2011	1136
2007	1197
2010	1255
2009	1426
2008	1427
1988	1427

Die relevanten Datenpunkte für dieses Experiment ist das Erscheinungsdatum des Spiels und die globalen Verkäufe des Spiels.

Dieser Datensatz ist auf Kaggle.de für jeden frei zugänglich.[37]

4.5.2 Storm

Apache Storm wurde von Grund auf so konzipiert, dass es mit jeder Programmiersprache verwendet werden kann. Nicht-JVM Spouts und Bolts kommunizieren mit Apache Storm über ein JSON-basiertes Protokoll über stdin/stdout. Adapter, die dieses Protokoll implementieren, gibt es für Ruby, Python, Javascript und Perl.[33]

Aufgrund der Kommunikation zwischen Apache Storm und den nicht-JVM Spouts und Bolts wurde das Experiment für Apache Storm in Java implementiert. Wenn weniger Nachrichten ausgetauscht werden, entsteht weniger Last, wodurch die Anwendung performanter sein sollte.

Das Experiment wurde mit der Streams API von Storm implementiert. Zunächst werden die Daten in Storm gelesen. In Storm benötigt es ein Kafkaspout, um Kafka-Themen zu abonnieren und die Nachrichten des Themas im System zu erhalten. Um Ereignisse in Kafka-Themen zu schreiben, benötigt man in Apache Storm einen Kafkabolt. Mithilfe eines benutzerdefinierten Translator kann der Zeitstempel, den der Kafka-Broker dem Ereignis in den Meta-Daten hinzugefügt hat, in Storm ausgelesen werden. Der Translator erhält den ConsumerRecord von Kafka und kann die Meta-Daten des Ereignisses in der überschriebenen Apply-Funktion auslesen (siehe: 3). Dies ist für die spätere Auswertung ein sehr wichtiger Datenpunkt. Zusätzlich wird beim Translator jedem Ereignis ein Zeitstempel hinzugefügt, der angibt, wann das Ereignis im Event-Streaming-Verarbeitungssystem eingetroffen ist.

```
1 class Translator implements RecordTranslator<String,
    String>{
2 @Override
3 public List<Object> apply(ConsumerRecord<String, String>
    arg0) {
4   //Append Log Time (arg0.timestamp());
5       return new Values(
6               arg0.timestamp(), arg0.value(),
7               String.valueOf((new Timestamp(System.
                    currentTimeMillis()).getTime()))
```

```
8           );
9   }
10 }
```

In Storm wird jedes Ereignis von Kafka als Tupel dargestellt. Mit einer Map Funktion wird der Stream von Tupeln zu einem Pairstream aus einem benutzerdefinierten Objekt und dem dazugehörigen Schlüssel gemappt. Das erstellte Objekt beinhaltet alle relevanten Datenpunkte, welche für die Analyse von Storm benötigt werden.

Die Klasse Game wird für alle Java Anwendungen verwendet (siehe: 1). Diese Klasse muss serialisiert werden können, damit die Game-Objekte in Bytes umgewandelt und in den Stream geschrieben werden können. Daher implementiert diese Klasse das Interface 'Serializable'.

```
1  public final class Game implements Serializable {
2          public Game(String name,
3                      String platform,
4                      int year,
5                      double globalSales,
6                      String timeSend,
7                      String timeProcessed,
8                      String appendLogTime,
9                      String writeTime,
10                     String timeInSps,
11                     double maxTimeProcessed) {
12                 this.name = name;
13                 this.platform = platform;
14                 this.year = year;
15                 this.globalSales = globalSales;
16                 this.timeSend = timeSend;
17                 this.timeProcessed = timeProcessed;
18                 this.appendLogTime = appendLogTime;
19                 this.writeTime = writeTime;
20                 this.timeInSps = timeInSps;
21                 this.maxTimeProcessed = maxTimeProcessed
                       ;
22         }
23 }
```

Nachdem die Objekte erstellt worden sind, werde diese aggregiert. Hierfür wird der Stream anhand des Jahres gruppiert und die Daten werden mithilfe einer benutzerdefinierten Reduce Funktion verarbeitet.

Die Reduce Funktion ist dafür zuständig, die globalen Verkäufe für jedes Jahr zu summieren und die Verarbeitungszeit zu berechnen. In der Reduce Funktion wird für jedes Ereignis die Verarbeitungszeit berechnet und die maximale Verarbeitungszeit wird für jeden Zustand gespeichert. Des Weiteren wird der neueste Zeitstempel vom Kafka-Broker für jeden Zustand benutzt, der zu dem Ergebnis des Zustandes beigetragen hat.

```
1  public class GameReducer<T> implements Reducer<Game> {
2  @Override
3  public Game apply(Game v1, Game v2) {
4          int compareTimeSend =
5           new Timestamp(Long.parseLong(v1.getAppendLogTime()))
6           .compareTo(new Timestamp(Long.parseLong(v2.
               getAppendLogTime()))));
7
8           String maxAppendLogTime = "";
9           double maxProcessingTime = 0;
10          double sumSales = v1.getGlobalSales() +  v2.
               getGlobalSales();
11          String timeInSps = "";
12         // v1 spaeter
13         if (compareTimeSend == 1){
14                 long processingTime =
15                 (new Timestamp(System.currentTimeMillis()).
                      getTime()) -
16                 (new Timestamp(Long.parseLong(v2.getTimeInSps()
                      )).getTime());
17                 maxAppendLogTime = v1.getAppendLogTime();
18                 if(v1.getMaxTimeProcessed() > processingTime){
19                         maxProcessingTime = v1.
                              getMaxProcessingTime();
20                         timeInSps = v1.getTimeInSps();
21                 }
22                 else{
23                 maxProcessingTime = (new Timestamp(System.
                      currentTimeMillis()).getTime())
24                         - (new Timestamp(Long.parseLong(v2.
                              getTimeInSps()))));
```

```
25                                  timeInSps = v2.getTimeInSps();
26
27                          }
28                  }
29          // v2 spaeter
30          else{
31                  long processingTime = (new Timestamp(System.
                        currentTimeMillis()).getTime())
32                          - (new Timestamp(Long.parseLong(v2.getTimeInSps
                            ()))).getTime());
33                  maxAppendLogTime = v2.getAppendLogTime();
34                  if(v1.getMaxTimeProcessed() > processingTime){
35                          maxProcessingTime = v1.getMaxProcessingTime();
36                          timeInSps = v1.getTimeInSps();
37
38                  }
39                  else{
40                          maxProcessingTime =      (new Timestamp(System.
                            currentTimeMillis()).getTime())
41                          -  (new Timestamp(Long.parseLong(v2.
                            getTimeInSps()))).getTime());
42                          timeInSps = v2.getTimeInSps();
43
44                  }
45           }
46           return new Game("reducer", "plat", v1.getYear(),
47                          sumSales,
48                          v1.getTimeSend(),
49                          String.valueOf(
50                          new Timestamp(System.currentTimeMillis()).
                            getTime()
51                          ),
52                          maxAppendLogTime, null,
53                          timeInSps, maxProcessingTime);
54          }
```

Nachdem die Daten verarbeitet und aggregiert worden sind, werden die Daten von Storm zurück in ein Kafka-Thema geschrieben. Bei Storm kann dem schreibenden Kafka-Bolt eine benutzerdefinierte Map-Funktion übergeben werden. Diese wurde so implementiert, dass die Funktion dem Ergebnis den Zeitstempel des Schreibens hinzufügt.

```
class Mapper<K, V> implements TupleToKafkaMapper<String,
    String>{
        @Override
        public String getKeyFromTuple(Tuple arg0) {
                return arg0.getValue(0).toString();
        }
        @Override
        public String getMessageFromTuple(Tuple arg0) {
                Game game = (Game) arg0.getValue(1);
                game.setWriteTime(
                        String.valueOf(
                         new Timestamp(System.
                             currentTimeMillis()).getTime
                             ())
                        );
                return game.toString();
        }
}
```

Der Storm-Job wurde 10x ausgeführt. Die transformierten und aggregierten Ereignisse wurden in Python ausgewertet und visualisiert. Die Ergebnisse werden im nächsten Kapitel dargestellt.

4.5.3 KSQLDB

KsqlDB erleichtert die Verarbeitung von kontinuierlichen Daten, da kein komplexes System benötigt wird, welches aus mehreren verteilten Systemen besteht. Dies vereinfacht erheblich die Bereitstellung und Verwaltung von Streaming-Abfragen in Ksql mit SQL-ähnlicher Syntax.

Die Daten werden wie in allen getesteten Systemen mit einem Pythonskript ausgelesen und in ein Kafka-Thema geschrieben. Mit KsqlDB ist es möglich, die Daten nur mit SQL-Befehlen zu erhalten, zu verarbeiten, zu aggregieren und diese automatisch in weitere Kafka-Themen zu schreiben.

Als Erstes wurde ein Stream in KsqlDB erstellt, dieser abonniert das Kafka-Thema, in welches die Daten vom Pythonskript geschrieben werden. Der Stream benötigt ein Schema der Daten und das Format der Daten.

```
client.ksql("""CREATE STREAM IF NOT EXISTS game (
    game STRUCT <name VARCHAR, platform VARCHAR,
                 year INT, global_sales DOUBLE, time_send
                     Varchar>
    ) WITH (KAFKA_TOPIC='games', VALUE_FORMAT='Avro')"""
        )
```

Anschließend wird die Append Log Time vom Kafka-Broker, die Rowtime und ein Unix Timestamp zu dem Stream hinzugefügt. Der Unix Timestamp gibt an, wann das Ereignis in KsqlDB eingetroffen ist.

```
client.ksql("""CREATE STREAM games_withRowtime WITH (
    FORMAT='JSON')
 AS SELECT
 game->year as year,
 game->global_sales as global_sales,
 TIMESTAMPTOSTRING(ROWTIME, 'yyyy-MM-dd HH:mm:ss.SSS')
     as appendLogTime,
 UNIX_TIMESTAMP() as time_in_sps
 FROM game""")
```

Nun hat man einen Stream mit allen relevanten Datenpunkten und Zeitstempeln. Nachfolgend werden die Daten gruppiert und aggregiert. Hier wird wie in Storm die Summe der globalen Verkäufe, die maximale Append Log Time sowie die maximale Verarbeitungszeit für jeden Zustand berechnet. Die Ereignisse werden in einer Tabelle gespeichert.

```
client.ksql("""CREATE TABLE test_output WITH (FORMAT='
    JSON')
 AS SELECT
  year,
        sum(global_sales) as sum_sales ,
        max(appendLogTime) as append_log_time,
        max(UNIX_TIMESTAMP() -time_in_sps) as
            processingTime,
        TIMESTAMPTOSTRING(UNIX_TIMESTAMP(),
          'yyyy-MM-dd HH:mm:ss.SSS') as write_time
        FROM games_withRowtime group by year""") |
```

Ein Stream oder eine Tabelle in KsqlDB erstellt automatisch ein Thema in Kafka, daher wird der Tabelle der Zeitstempel des Schreibens hinzugefügt. Es entfällt das explizite Schreiben der Daten in ein Kafka Thema.

Der KsqlDB-Job wurde 10x ausgeführt. Die transformierten und aggregierten Ereignisse wurden in Python ausgewertet und visualisiert. Die Ergebnisse werden im nächsten Kapitel dargestellt.

4.5.4 Kafka-Streams

Mit Kafka-Streams ist es möglich, die Daten in Kafka-Themen direkt zu verarbeiten. Hierfür wurde eine Anwendung in Java implementiert, mit welcher es möglich ist, die Daten in Kafka-Streams aus einem Thema auszulesen, diese zu aggregieren, die verschiedenen Zeitstempel hinzuzufügen und die Daten wieder in ein Kafka-Thema zu schreiben.

In Kafka-Streams ist es möglich, die Meta-Daten mit der Hilfe einer Klasse auszulesen, welche das Interface 'Transformer' implementiert. Anschließend kann man in dieser Klasse eine Funktion implementieren, welche die Ereignisse von Kafka-Streams transformiert. Innerhalb dieser Transformationen können die Meta-Daten der Ereignisse ausgelesen werden und man erhält die Append Log Time vom Kafka-Broker.

Die Ereignisse in Kafka-Streams werden als JSON entgegengenommen und müssen, nachdem die Meta-Daten hinzugefügt worden sind, gemappt werden, sodass mit den Daten gearbeitet werden kann. Hierfür wurde wieder die benutzerdefinierte Klasse Game implementiert, welche die Daten widerspiegelt (siehe: 1). Der JSON-String wird zu einem Objekt gemappt und beim Mappen wird dem Ereignis ein Zeitstempel hinzugefügt, welcher angibt, wann das Ereignis im Event-Streaming-Verarbeitungssystem angekommen ist.

Nachdem die Daten vorbereitet worden sind, werden diese nach ihrem Jahr gruppiert und die Summe der globalen Verkäufe wird mit der maximalen Verarbeitungszeit je Zustand berechnet. Zudem wird die maximale Append Log Time für jeden Zustand dem Ergebnis angereichert. Um dies umzusetzen, wurde eine benutzerdefinierte Reduce Funktion implementiert, welche dieselbe Logik aufweist wie die Reduce Funktion von Storm. Lediglich wird bei Kafka-Streams nicht die GroupBy Funktion aufgerufen, sondern die Daten werden mit der Funktion "ReduceByKey" übergeben. Diese Funktion gruppiert

die Daten anhand des Schlüssels und ruft anschließend die benutzerdefinierte Reduce Funktion auf (siehe: 1).

Der Kafka-Streams-Job wurde 10x ausgeführt. Die transformierten und aggregierten Ereignisse wurden in Python ausgewertet und visualisiert. Die Ergebnisse werden im nächsten Kapitel dargestellt.

Apache Spark

Apache Spark ist in Scala geschrieben und ermöglicht die Programmierung in Python, Scala, R und Java. Für Python gibt es zum Beispiel das Interface PySpark. PySpark benötigt jedoch einen Proxy, der in der Lage ist, den Code aus Python zu übernehmen, diesen an die JVM weiterzuleiten und die Ergebnisse bei Bedarf zurückzuholen. Die dafür in PySpark verwendete Proxy-Schicht ist die Py4J-Bibliothek. Um eine einheitliche Bewertung der ausgewählten Streaming-Systeme zu erhalten, wurde der Apache Spark Job in Java implementiert. Hierdurch wird keine Proxy-Schicht benötigt.

In Spark ist es möglich, mit zwei verschiedenen Abstraktionen Spark Streaming und Spark Structured Streaming kontinuierliche Datenströme zu verarbeiten.

Spark Streaming

Der implementierte Job verwendet die Abstraktionsebene Spark Streaming. Dieser Job wurde implementiert, da in Apache Beam der Spark Structured Streaming 'Runner' keinen Streaming-Mode unterstützt.3.6.2

Als Erstes wurden die Daten mithilfe von einem Python Skript gelesen und in ein Kafka-Thema geschrieben.

Mit der Spark Streaming API ist es möglich, die Meta-Daten der Ereignisse in einer Map-Funktion zu erhalten. Daher wurde eine Map-Funktion implementiert, welche die Daten mithilfe eines ObjectMappers ausliest und ein benutzerdefiniertes Klassenobjekt erstellt (siehe: 1). Innerhalb der Map-Funktion wird das Objekt mit den Meta-Daten ausgelesen, sodass man den Zeitstempel, die Append Log Time vom Message-Broker erhält. Zudem wird ein weitere Zeitstempel hinzugefügt, welcher angibt, wann das Ereignis im Event-Streaming-Verarbeitungssystem angekommen ist.

Nachdem die empfangenen Ereignisse mit den Zeitstempeln angereichert worden sind, können diese verarbeitet werden. Hierfür wird eine benutzerdefinierte updateStateByKey Funktion implementiert. Diese Funktion bekommt als Parameter eine Liste von den gruppierten Schlüsselpaaren und einem Zustand. Innerhalb dieser Funktion wird berechnet, welches Ereignis die maximale Verarbeitungszeit für jeden Schlüssel benötigt. Zudem wird der Zeitstempel, die Append Log Time, ausfindig gemacht, welcher als letztes zu einem Ergebnis einen Beitrag geleistet hat und die Summe der globalen Verkäufe werden pro Schlüssel aufsummiert. Am Ende wird das Ergebnis wieder mit einem Zeitstempel angereichert, um anzugeben, wann das Ergebnis zurück in ein Kafka-Thema geschrieben worden ist.

Der Spark-Streaming-Job wurde 10x ausgeführt. Die transformierten und aggregierten Ereignisse wurden in Python ausgewertet und visualisiert. Die Ergebnisse werden im nächsten Kapitel dargestellt.

Spark Structured Streaming

Der implementierte Job verwendet die neuere Abstraktionsebene Spark Structured Streaming. Zunächst wurden wieder die Daten mithilfe von einem Pythonskript ausgelesen und in ein Kafka-Thema eingespeist.

Anschließend werden die Daten in Apache Spark gelesen und mit der Hilfe einer Map-Funktion zu einem benutzerdefinierten Objekt gemappt. Dieses Objekt beinhaltet alle relevanten Datenpunkte und den Zeitstempel, welcher vom Kafka-Broker in den Meta-Daten mitgesendet worden ist. Zusätzlich wird in der Map-Funktion ein weiterer Zeitstempel definiert, welcher angibt, wann das Ereignis in Apache Spark gelesen worden ist.

Nachdem die Zeitstempel zu den Daten angereicht worden sind, werden die Daten verarbeitet. Um die Verarbeitungszeit für jedes Ereignis zu berechnen, wurde in Apache Spark eine benutzerdefinierte MapGroupsWithState Funktion implementiert.

In dieser Funktion wird die maximale Verarbeitungszeit für jeden Zustand berechnet und es werden die Daten aggregiert. Die Summe der globalen Verkäufe sowie der neueste Zeitstempel, welcher vom Kafka-Broker erhalten worden ist, wird für jeden Zustand berechnet bzw. ausgewählt. Die Funktion erhält jedoch nicht wie bei der Reduce Funktion

von Storm einen Zustand und ein weiteres Ereignis, welches verarbeitet werden soll, sondern einen Batch von Ereignissen und einen Zustand. Zunächst werden in der Funktion iterativ alle Ereignisse durchlaufen und die gewünschten Attribute werden aggregiert und als Zustand gespeichert.

Bevor die MapGroupsWithState Funktion aufgerufen wird, werden die Daten gruppiert. Nachdem die Daten verarbeitet worden sind, wird mit einer weiteren Map-Funktion der Zeitstempel des Schreibens hinzugefügt. Anschließend werden die Daten vom Spark Job in ein Kafka-Thema geschrieben.

Der Spark-Job wurde 10x ausgeführt. Die transformierten und aggregierten Ereignisse wurden in Python ausgewertet und visualisiert. Die Ergebnisse werden im nächsten Kapitel dargestellt.

4.5.5 Apache Flink Java

Um Apache Flink zu testen, wurde die DataStream-API von Flink benutzt und die Anwendung wurde in Java implementiert.

Bei dieser Anwendung wurden wieder Daten mithilfe von einem Pythonskript in ein Kafka-Thema geschrieben. Die Daten werden in Flink mit dem Kafka-Connector gelesen, der Kafka-Connector wird als Abhängigkeit dem Flink-Job übergeben. Es ist möglich, der Kafka-Quelle, welche in Flink initialisiert wird, einen Deserializer zu übergeben. Dieser konvertiert die ankommenden JSON-Strings zu Objekten. Hierfür wurde wieder eine benutzerdefinierte Klasse implementiert, welche die relevanten Datenpunkte beinhaltet (siehe: 1). Mit dem Deserializer ist es zudem möglich, die Append Log Time (siehe: 11) vom Kafka-Broker zu erhalten:

```
1
2 public class GameDeserialization
3                 implements
                      KafkaRecordDeserializationSchema<Game
                      > {
4         @Override
5         public void deserialize(ConsumerRecord<byte[],
              byte[]> arg0, Collector<Game> arg1)
6                       throws IOException {
7                 Game data;
8                 try {
```

```
9                               objectMapper.configure(
                                    DeserializationFeature.
                                    FAIL_ON_UNKNOWN_PROPERTIES,
                                    false);
10                              data = objectMapper.treeToValue(
                                    objectMapper.readTree(arg0.
                                    value()), Game.class);
11                              data.setAppendLogTime(String.
                                    valueOf(arg0.timestamp()));
12                              data.setTimeInSps(String.valueOf
                                    ((new Timestamp(System.
                                    currentTimeMillis()).getTime
                                    ())));
13                      } catch (Exception e) {
14                              throw new SerializationException
                                    (e);
15                      }
16                      arg1.collect(data);
17              }
18      }
```

Anschließend werden die Ereignisse nach dem Jahr gruppiert und die Summe der globalen Verkäufe, die neueste Append Log Time sowie die maximale Verarbeitungszeit wird für jeden Zustand berechnet. Dies geschieht in Flink mit derselben Reduce Funktion wie in Storm (siehe: 1).

Wenn die Daten fertig verarbeitet worden sind, werden die Daten mithilfe von dem Kafka-Connector und einem benutzerdefinierten Serialisierer in ein Kafka-Thema geschrieben. Im Serialisierer wird der Zeitpunkt des Schreibens festgehalten.

Der Flink-Java-Job wurde 10x ausgeführt. Die transformierten und aggregierten Ereignisse wurden in Python ausgewertet und visualisiert. Die Ergebnisse werden im nächsten Kapitel dargestellt.

4.5.6 Apache Flink Python

Apache Flink ist ein in Java geschriebenes Framework. PyFlink ist eine Schnittstelle zu Flink. Der Python-Code wird mit Py4j in Java kompiliert. Der kompilierte Code kann an den Flink-Job-Manager übermittelt werden, um auf dem Flink-Cluster ausgeführt zu

werden. Der Code kann sowohl mit der Python-VM als auch mit der Flink-VM im Cluster interagieren, auf dieser Weise kann weiterhin mit Python-Bibliotheken gearbeitet werden. Es existieren zwei verschiedene APIs für PyFlink. Die PyFlink Table API ermöglicht es, leistungsfähige relationale Abfragen zu schreiben, die der Verwendung von SQL oder der Arbeit mit tabellarischen Daten in Python ähneln. Des Weiteren gibt es die PyFlink DataStream API, mit dieser können komplexe Streaming Jobs mit einem Zustand und Zeit erstellt werden.

Da der Python-Code mit Py4j kompiliert wird, wurde überprüft, wie viel Zeit für diese Kompilierung benötigt wird. Daher wurde der Streaming-Job auch in PyFlink implementiert und anaylisiert.

Als Erstes wurden wieder die Daten mithilfe von einem Pythonskript in ein Kafka-Thema geschrieben. In Pyflink wurden die Daten zunächst mit dem 'flink-sql-connector-kafka' gelesen. Dieser Konnektor wird benötigt, um in PyFlink mit Kafka zu interagieren. Anschließend wurden den Ereignissen mithilfe einer Map-Funktion ein Zeitstempel hinzugefügt, welcher angibt, wann das Ereignis im Stream Verarbeitungssystem angekommen ist.

Um die Append Log Time vom Kafka-Broker zu erhalten, gibt es in PyFlink zwei Möglichkeiten. Die erste Möglichkeit ist es, den Stream in eine Tabelle zu konvertieren. Daraufhin ist es möglich, mit dem Attribut Rowtime die Append Log Time vom Kafka-Broker zu erhalten.

```
1  env = StreamExecutionEnvironment.
       get_execution_environment()
2  t_env = StreamTableEnvironment.create(
       stream_execution_environment=env)
3  ...
4  tbl = t_env.from_data_stream(ds,
5            col('name'), col('platform'), col('year'),
6            col('global_sales'), col('time_send'), col('
                 append_log_time').rowtime,
7            col('time_in_sps'), col('write_time'), col('
                 processed_time'),
8            col('max_processing_time'))
9  .alias("name", "platform","year","global_sales","
       time_send",
10           "append_log_time","time_in_sps","write_time",
11           "processed_time", "max_processing_time")
```

Für die Alternative muss eine Process Funktion implementiert werden. In dieser ist es möglich, auf den Kontext zuzugreifen. Der Kontext weist ein Attribut 'timestamp' auf, dieses beinhaltet die Rowtime, welche vom Kafka-Broker in den Meta-Daten an Flink übergeben worden ist.

```
class MyProcessFunction(KeyedProcessFunction):
    def process_element(self, value, ctx: 'KeyedProcessFunction.Context'):
      yield Row(
                            str(value[0]),
                            str(value[1]),
                            str(value[2]),
                            str(value[3]),
                            str(value[4]),
                            str(ctx.timestamp() * 1000), //Append Log Time
                            str(value[6]),
                            str(value[7]),
                            str(value[8]),
                            "0.0"
                           )
```

Um nur mit der DataStream API zu arbeiten, wurde die zweite Alternative implementiert und verwendet.

Nachdem die Zeitstempel hinzugefügt worden sind, wurden die Ereignisse gruppiert und aggregiert. Um dies umzusetzen, wurde wieder eine Reduce Funktion implementiert, welche dieselbe Logik aufweist wie die Reduce Funktion in Apache Storm (siehe: 1). Mit einer weiteren Map-Funktion wird den Ereignissen, bevor diese zurück in ein Kafka-Thema geschrieben werden, mit einem Zeitstempel angereichert. Dieser gibt an, wann das Stream-Verarbeitungssystem die Daten in Kafka geschrieben hat.

Der PyFlink-Job wurde 10x ausgeführt. Die transformierten und aggregierten Ereignisse wurden in Python ausgewertet und visualisiert. Die Ergebnisse werden im nächsten Kapitel dargestellt.

4.5.7 Apache Beam

Für Apache Beam wurde eine Pipeline implementiert, welche dieselben Transformationen und Aggregationen enthält wie in den implementierten Streaming-Jobs der vorgestellten Event-Streaming-Verarbeitungssysteme.

Die Pipeline erhält Daten aus einem Kafka Thema. Anschließend werden die empfangenen Daten mithilfe einer ParDo Funktion in Beam ausgelesen und die Zeitstempel sowie die Append Log Time vom Kafka-Broker werden zu den Daten angereichert.

In Apache Beam wird standardmäßig das 'Globale Window' mit einem Trigger, welcher nach der Watermark abläuft, benutzt. Es ist daher nicht möglich in Beam mit kontinuierlichen Daten zu arbeiten ohne ein spezifisches Window zu definieren. Es wurde ein Fixed Window mit einer Dauer von 600 Sekunden definiert und ein Trigger, welcher die Aggregationen nach jedem Element auslöst. In Apache Beam ist es möglich, die Ergebnisse der unterschiedlichen Windows zu akkumulieren oder wegzuwerfen. Da die anderen Event-Streaming-Verarbeitungssysteme die aggregierten Daten akkumulieren, wurde dies auch in Beam definiert.

Anschließend werden die Daten mithilfe einer CombineFn Funktion verarbeitet. Hier wird die maximale Verarbeitungszeit für jeden Zustand berechnet und die Daten werden aggregiert. Die Funktion hat nahezu denselben Aufbau wie die Reduce-Funktion in Apache Storm (siehe: 1).

Es ist außerdem möglich, die Funktionalität mit der Gruppierung, Aggregation und Berechnung der Verarbeitungszeit mit einer zustandsbehafteten DoFn Funktion umzusetzen. Dies wird jedoch noch nicht vom Apache Spark Runner unterstützt.3.6.2

Am Ende fließen die Daten zurück in ein Kafka-Thema, wo diese von einem Pythonskript gelesen und ausgewertet werden.

4.5.8 Apache Beam mit Flink

Die Pipeline wurde mit dem Apache Flink Runner von Apache Beam ausgeführt.

Der Streaming-Job wurde 10x ausgeführt und die Ereignisse wurden visualisiert. Siehe: 5.8

4.5.9 Apache Beam mit Spark

Zudem wurde dieselbe Pipeline mit dem Apache Spark Runner von Apache Beam ausgeführt. Da Apache Beam noch keinen Streaming-Mode für den Spark Structured Streaming Runner unterstützt, wurde der klassische Spark DStream Runner benutzt.

Während der Analyse der Ergebnisse des Spark Runners sind mehrere Auffälligkeiten hervorgestochen. Die Processing-Time-Latenz stieg mit steigender Dauer des Jobs kontinuierlich an. Nach einer gründlichen Analyse der Pipeline für den Spark Runner ist aufgefallen, dass der Spark Runner bisher verarbeitete Ereignisse nicht als verarbeitet markiert. Das führte dazu, dass diese in jedem Durchgang wieder neu verarbeitet werden. Die Processing-Time-Latenz wird als die Zeit zwischen der Zeit, in der das Ereignis im System angekommen ist und der Zeit, wann es fertig verarbeitet worden ist, gemessen. Daher steigt die Verarbeitungszeit in jedem Batch, da die maximale Verarbeitungszeit in jedem Durchgang für jedes Event neu berechnet wird. Dadurch, dass für jedes Event der Zeitstempel benutzt wird, wann das Ereignis im System angekommen ist, ist das erste Event immer automatisch das Event mit der maximalen Verarbeitungszeit, welches kontinuierlich mit vergangener Zeit ansteigt.

Die Processing-Time-Latenz musste in Apache Beam mit dem Spark Runner daher anders gemessen werden. Es wurde die Verarbeitungszeit zurückgegeben, die das letzte Ereignis in einem Durchgang benötigt hat.

Der Streaming-Job wurde in Beam mit dem Spark Runner 10x ausgeführt und die Ergebnisse wurden in Python visualisiert. Die Ergebnisse sind in 5.9 zu sehen.

5 Diskussion

Im folgenden Kapitel werden die Ergebnisse des Experimentes vorgestellt und die Forschungsfragen beantwortet.

5.1 Ergebnisse der Event-Streaming-Verarbeitungssysteme

In diesem Abschnitt werden die Ergebnisse des Experimentes vorgestellt.

Apache Storm

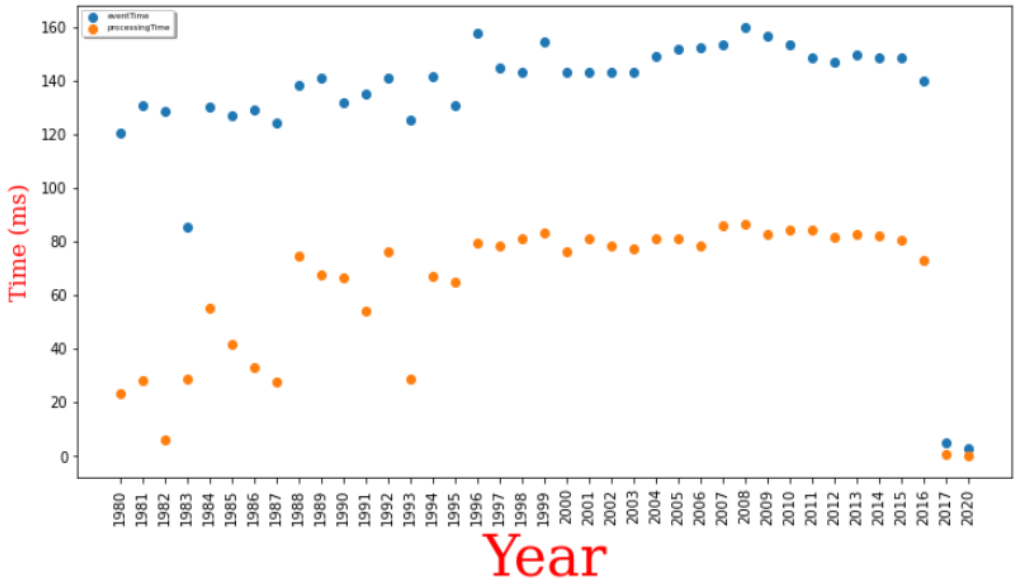

Abbildung 5.1: Storm Benchmark

Die Event-Time-Latenz wird in der Abbildung (siehe: 5.1) durch die blauen Punkte dargestellt. Die orangenen Punkte geben an, wie lange die Verarbeitung maximal für einen

Zustand gedauert haben. Wie in 5.1 zu sehen ist, benötigt Storm auf der verwendeten Hardware nur bis zu 160ms pro Zustand für das Senden, Transformieren und Aggregieren der Daten. Die Verarbeitungszeit ist extrem niedrig. Die maximale Verarbeitungszeit liegt im Jahr 2008 und 2009 bei ca. 80ms. Dies könnte daran liegen, dass die meisten Ereignisse in dem Datensatz von Spielen aus dem Jahr 2008 und 2009 sind. In diesen beiden Jahren sind 1426 bzw. 1427 Ereignisse vorhanden. Im Jahr 2020 und 2017, in denen die gemessenen Latenzzeiten extrem niedrig sind, sind nur extrem wenige Ereignisse vorhanden.

Storm ist in der Lage, in nahezu Echtzeit Ereignisse zu empfangen, diese zu verarbeiten, zu aggregieren und zurückzugeben.

Komplexität Apache Storm

Es ist möglich, Apache Storm in mehreren Programmiersprachen zu implementieren. Die zur Auswahl stehenden Programmiersprachen sind Python, Java oder Ruby. Dies reduziert die Komplexität, da der Entwickler mehrere Sprachen zur Auswahl hat. Es sollte jedoch beachtet werden, dass Anwendungen, welche nicht mit Java implementiert worden ist, über ein JSON-basiertes Protokoll mit Apache Storm kommunizieren. Dadurch können die Latenzzeiten ansteigen.

Der Aufwand, um den Streaming-Job in Apache Storm zu implementieren, war überschaubar.

Insgesamt benötigt der Job nur 191 Programmierzeilen. Die Pipeline benötigt nur 21 Programmierzeilen. Die Reduce Funktion weist 34 Programmierzeilen auf und die Map-Funktion, sowie der Translator, mit welchem die Meta-Daten der Ereignisse ausgelesen werden können, haben 21 Programmierzeilen. 89 Programmierzeilen werden für die benutzerdefinierte Klasse benötigt.

Die Implementierung eines Jobs in Apache Storm mit Java ist durch die vielen Informationen und Tutorials, welche im Internet zur Verfügung stehen, nicht sehr komplex. Es existiert eine große Apache Stom Community. [32] Spezifische Funktionen zu implementieren, wie z.B. den RecordTranslator, um den Zeitstempel vom Message-Broker zu erhalten, waren zudem unkompliziert und schnell umgesetzt. Es besteht ein geringer Zeitaufwand, um in die Konzepte der Spouts, Bolts und Topologien einzusteigen.

KsqlDB

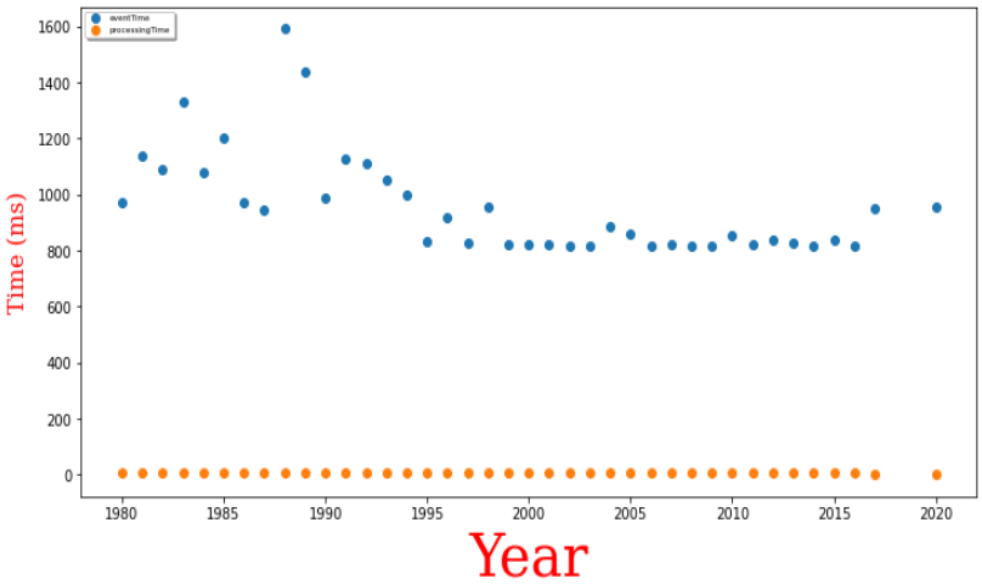

Abbildung 5.2: KsqlDB Benchmark

Die Verarbeitungszeit bei KSQLDB ist wie in 5.2 zu sehen, sehr gering. Diese beträgt nur wenige Millisekunden. Jedoch ist zu beachten, dass die Event-Time Latenz in KsqlDB ein Vielfaches der Verarbeitungszeit ist. Das Hinzufügen der Meta-Daten sowie das Lesen und Schreiben der Daten in KsqlDB ist nicht sehr performant. Diese Schritte benötigen bei gewissen Zuständen bis zu 1.6 Sekunden.

Komplexität KsqlDB

KsqlDB erleichtert die Event-Stream-Verarbeitung, da kein komplexes System benötigt wird, welches aus mehreren verteilten Systemen besteht. Zudem benötigt es für KsqlDB keine Kenntnisse über eine Programmiersprache, wie zum Beispiel Java oder Python. Durch die Verwendung von SQL-ähnlicher Syntax ist es sehr einfach, einen Streaming-Job mit KsqlDB zu erstellen, sofern Vorkenntnisse in SQL vorhanden sind. Dies erleichtert den Einstieg in die Event-Stream-Verarbeitung. In KsqlDB ist es mit drei SQL-Statements möglich, diesen Job zu implementieren. 1. Stream erstellen (Kafka Thema als Eingabe) 2. 'Append log time' auslesen und Zeitstempel hinzufügen (Zeit im System) 3. Aggregation des Streams und hinzufügen eines Zeitstempel (Zeit geschrieben)

Kafka-Streams

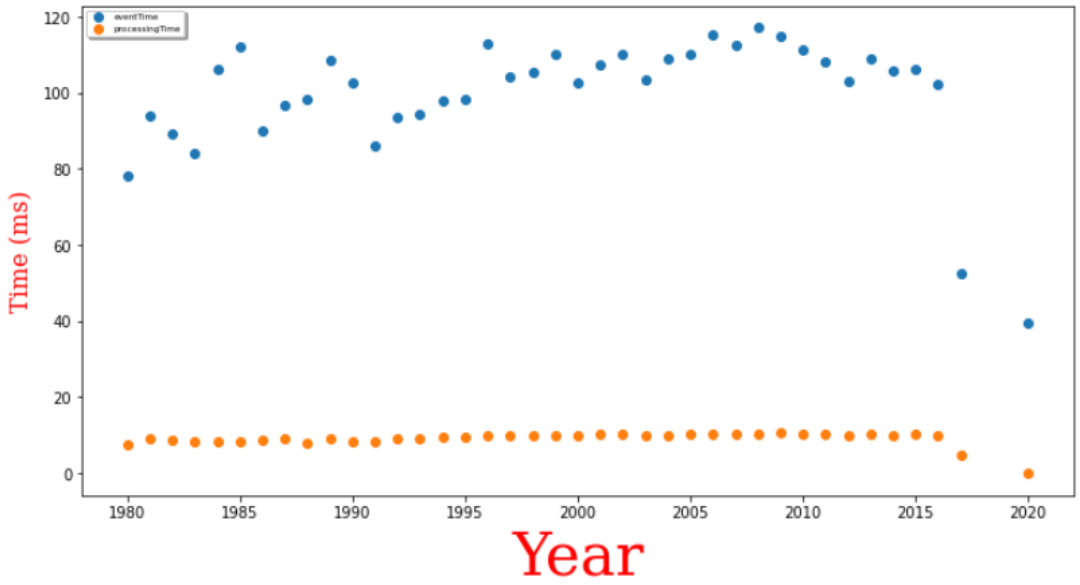

Abbildung 5.3: Kafka-Streams Benchmark

Die Event-Time Latenz in Kafka-Streams ist sehr gering, die maximale Event-Time Latenz beträgt 120ms. Siehe: 5.3 Im Jahr 2008 und 2009 ist die Event-Time Latenz wieder am höchsten, da in diesen Jahren die meisten Ereignisse verarbeitet werden. Die Verarbeitungszeit ist extrem gering. Nachdem die Daten gelesen worden sind, berechnet Kafka-Streams die Verarbeitungszeit und aggregiert die Daten in unter 20ms für jeden Zustand.

Komplexität Kafka-Streams

Kafka-Streams kann mit Java oder Scala entwickelt werden. Die Implementierung von Kafka Streams kann herausfordernd und aufwändig sein.

Die Pipeline, welche die gesamte Verarbeitung steuert, weist in Kafka-Streams 57 Programmierzeilen auf. Den größten Anteil an der Entwicklung des Jobs für Kafka-Streams haben die Serialisierer, Deserialisierer, der Transfomer, um die Meta-Daten der Ereignisse zu erhalten und die benutzerdefinierte Klasse. Diese weisen zusammen 164 Programmierzeilen auf. Für die Reduce Funktion, welche für die Verarbeitung zuständig ist, werden lediglich 22 Programmierzeilen benötigt.

Es erfordert ein solides Verständnis von Apache Kafka und es müssen Kenntnisse in Java oder Scala und Maven oder Gradle zum Bauen der Jar-Dateien vorhanden sein. Des Weiteren sind spezifische Funktion wie den getTopicDetailsTransformer, um den Zeitstempel vom Message-Broker zu erhalten und die Reduce Funktion sehr einfach zu implementieren.

Spark Streaming

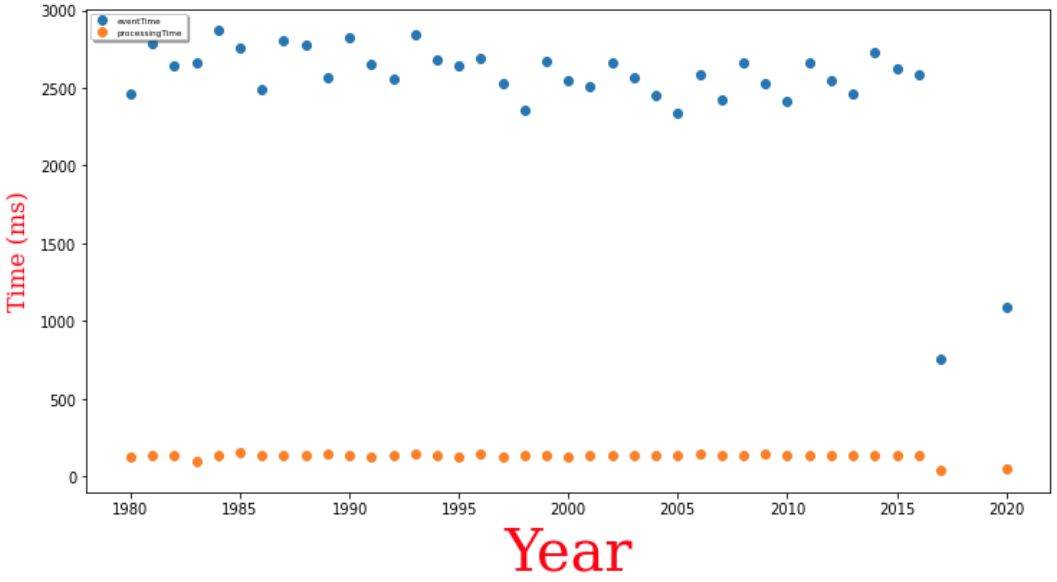

Abbildung 5.4: Spark Streaming Benchmark

Wie in der Abbildung 5.4 zu sehen ist, benötigt der Streaming-Job pro Zustand ca. 3 Sekunden, um von Spark gelesen, verarbeitet und geschrieben zu werden. Im Jahr 2020 und 2017, in denen nur wenige Ereignisse zu einem Zustand gehören, liegt die Event-Time-Latenz zwischen 0.5 und 1 Sekunde und ist erheblich geringer als die Latenzzeiten für die restlichen Zustände.

Die Verarbeitungszeit ist sehr gering und beträgt pro Zustand für ein Ereignis zwischen 50 und 200ms.

Komplexität Spark Streaming

Spark Streaming kann mit Java, Scala oder Python entwickelt werden. Informationen und Beispiele zu Implementierungen sind in einer großen Anzahl öffentlich zugänglich, sodass der Einstieg in die Implementierung von Spark Streaming Anwendungen erleichtert wird. Man muss beim Suchen nach Informationen jedoch aufpassen, da oft über die Structured Streaming API geschrieben und nicht über die ältere Streaming-API. Es existiert eine große Spark Community. [32]

Der Aufwand der Implementierung für Spark Streaming mit Java ist begrenzt, insgesamt werden 240 Programmierzeilen für die Implementierung des Jobs benötigt. Die Pipeline benötigt 45 Programmierzeilen. Lediglich die Implementierung der updateStateByKey Funktion ist nicht trivial. Diese Funktion weist alleine 80 Programmierzeilen auf. Die benutzerdefinierte Klasse sowie eine Funktion, um die Ergebnisse in ein Kafka-Thema zu schreiben haben 115 Programmierzeilen.

Die Notwendigkeit, die richtige Konfiguration von Spark-Parametern wie zur Speicherverwaltung oder dem State Management zu finden, um eine optimale Leistung zu erzielen, steigert die Komplexität. Spark Streaming ist eine komplexe Technologie, die von vielen Faktoren abhängt und eine gründliche Kenntnis von Spark und der zugrunde liegenden Systemarchitektur erfordert, um eine effektive Verarbeitung von großen Datenmengen zu gewährleisten.

Spark Structured Streaming

Wie in 5.5 zu sehen ist, benötigt der Spark Job im Vergleich zu anderen Streaming Systemen sehr lange. Die Event-Time-Latenz beträgt zwischen 3 und 8 Sekunden. Die Verarbeitungszeit liegt zwischen 0.2 und 5 Sekunden. Spark verwendet Mikro-Batches als Technik, um die Daten zu verarbeiten. Bei der Mikro-Batch-Verarbeitung werden Daten in kleinen Gruppen, sogenannte Mikro-Batches unterteilt, um diese anschließend zu verarbeiten. Die Mikro-Batch-Verarbeitung ist eine Variante der traditionellen Batchverarbeitung, bei der die Datenverarbeitung häufiger stattfindet.

Durch das Warten auf einen vollständigen Mikro Batch können höhere Latenzzeiten entstehen. Im Gegensatz zur Mikro Batch Verarbeitung werden in anderen Systemen Ereignisse nacheinander verarbeitet. In Bezug auf die Latenzzeit ist die 'Record-by-Record'-Verarbeitung eindeutig performanter.

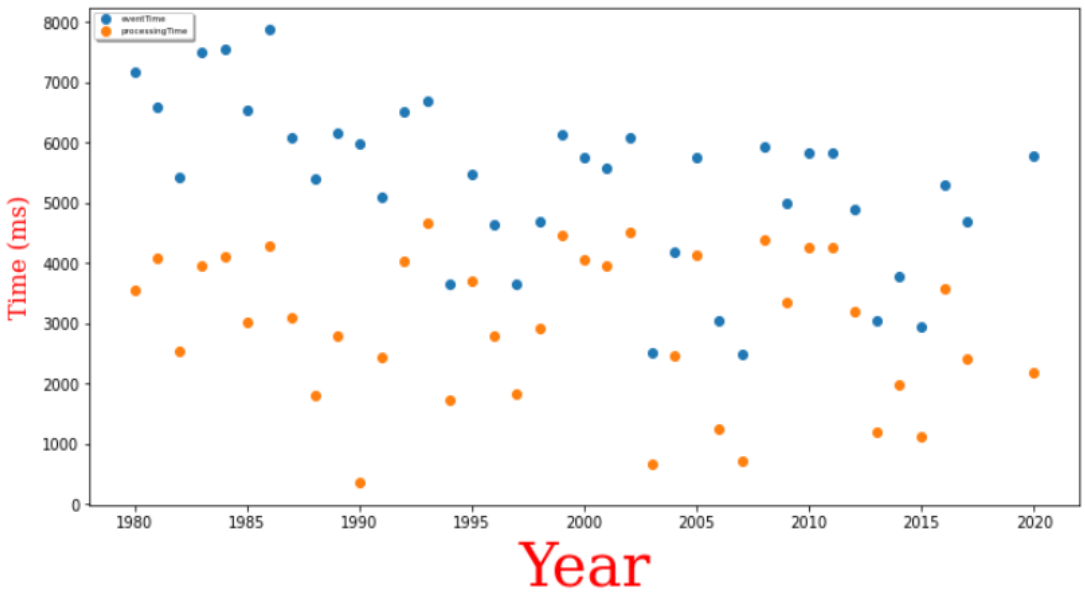

Abbildung 5.5: Spark Benchmark

So können in Spark gewissen Parameter konfiguriert werden, um die Performance der Verarbeitung zu verbessern. Es ist zum Beispiel möglich, die 'Mikro Batch Size' zu konfigurieren, welche bestimmt bei welcher Anzahl von Ereignissen der Mikro Batch verarbeitet werden soll. Je öfter ein Mikro Batch verarbeitet wird, desto geringer fallen die Latenzzeiten aus. In diesem Experiment wurden die standardmäßigen Einstellungen der Systeme verwendet.

Komplexität Spark Structured Streaming

Spark Structured Streaming bietet Schnittstellen zu Python, Java oder Scala.

Die Implementierung des Jobs mit Spark Structured Streaming mit Java ist aufwändig, es werden insgesamt 433 Programmierzeilen benötigt. Die Pipeline hat alleine 78 Programmierzeilen, die MapGroupsWithStateFunction 80 Programmierzeilen und die restlichen benutzerdefinierten Klassen weisen 275 Programmierzeilen auf. Für die MapGroupsWithStateFunction werden verschieden Objekte benötigt. Ein Eingabeobjekt, ein Objekt, welches die Aktualisierung widerspiegelt und ein Ausgabeobjekt. Daher existieren in Spark Structred Streaming drei verschiedene benutzerdefinierte Klassen.

Die Implementierung des Jobs war aufgrund der vielen Informationen und Beispiele, welche im Internet zur Verfügung stehen, einfach. Wie oben schon erwähnt, existiert eine große Spark Community.[32]

Lediglich die benutzerdefinierte MapGroupsWithStateFunction und der Erhalt der Meta-Daten aus den Ereignissen, um den Zeitstempel des Message-Brokers auszulesen, waren nicht trivial. Spark Strucured Streaming erfordert ein gewisses Maß an Erfahrung und Kenntnissen in der Datenverarbeitung und Programmierung, um das System effektiv nutzen zu können.

Apache Flink

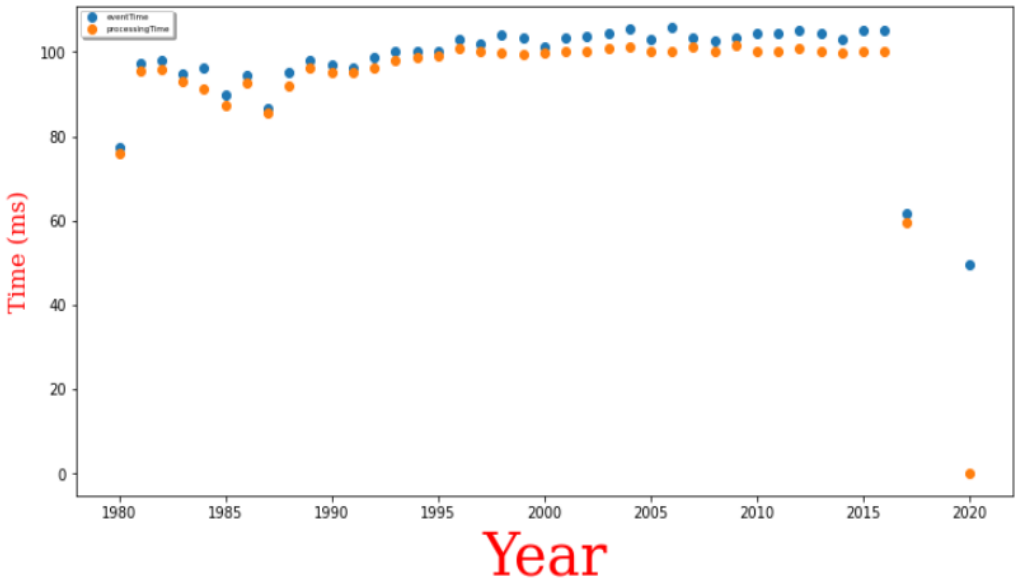

Abbildung 5.6: Flink Java Benchmark

Die Event-Time Latenz ist nahezu identisch mit der Verarbeitungszeit. Flink liest die Daten, verarbeitet diese und gibt diese wieder an Kafka in nahezu Echtzeit zurück.

Wie in Abbildung: 5.6 zu sehen ist, weist Flink eine Event-Time Latenz von maximal 105 ms auf. Die Verarbeitungszeit liegt maximal bei 103ms. Flink benötigt sehr wenig Zeit zum Lesen und Schreiben der Daten. Dies geschieht in unter 5ms.

Apache Flink Python (PyFlink)

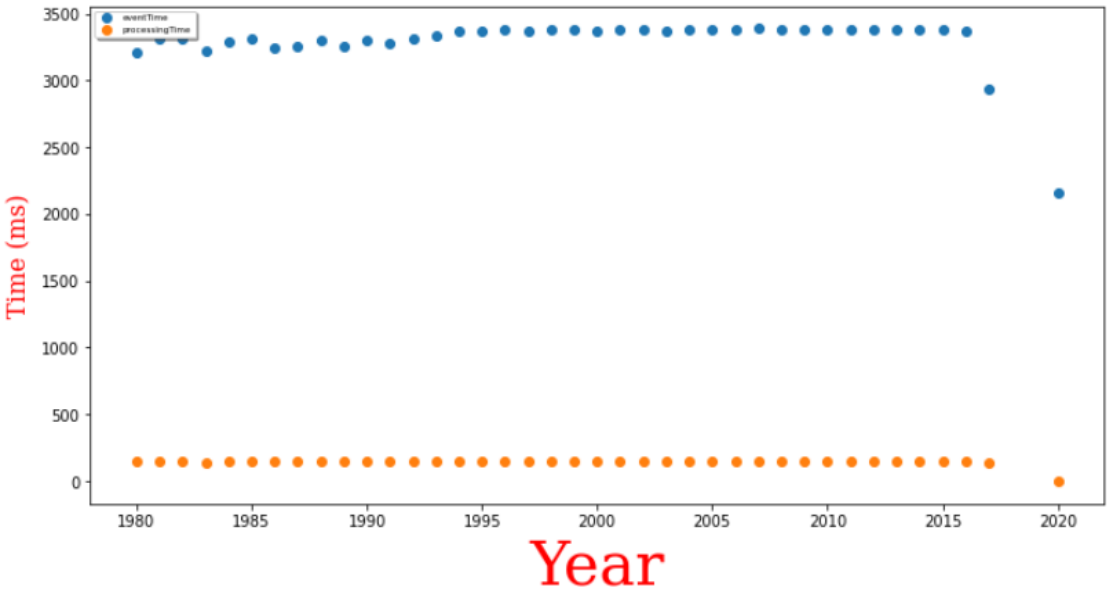

Abbildung 5.7: Flink Python Benchmark

In der Grafik: 5.7 ist zu sehen, dass der in PyFlink implementierte Job ein Vielfaches von der Zeit braucht,im Vergleich mit demselben Job, der in Java implementiert worden ist. Siehe: 5.6

Die Event-Time-Latenz liegt bei Python bei ca. 3 Sekunden. In Java benötigt der gesamte Job nur bis zu 110ms. Dies ist ein eklatanter Unterschied. Die Übersetzung von Python zu einer JVM Sprache kostet Ressourcen, welche hier deutlich werden. In Python benötigt der Job auch länger, um die Ereignisse zu verarbeiten. Der gesamte Job dauert in Python bis zu 30x länger, die Verarbeitungszeit ist um ein zehnfaches bis fünfzehnfaches länger.

Komplexität Apache Flink

Apache Flink lässt sich in Java, Scala oder Python entwickeln.

Die Komplexität mit PyFlink ist wesentlich geringer als die Implementation in Java.

Die gesamte Pipeline weist in PyFlink nur 76 Programmierzeilen aus. Dieselbe Pipeline in Java benötigt alleine für die Reduce Funktion, die Konfiguration der Pipeline sowie des Serialisierer und Deserialisierer mehr Programmierzeilen. Bei Java kommt noch die

benutzerdefinierte Klasse hinzu. Insgesamt benötigt Apache Flink mit Java 194 Programmierzeilen.

Nicht nur die Programmierzeilen sind in Python geringer, sondern auch der Aufwand, um Transformationen oder Aggregation durchzuführen. Python ist einfach und unkompliziert.[40]

Die Komplexität von Apache Flink ist hoch, es erfordert Kenntnisse in verschiedenen Bereichen und erfordert eine sorgfältige Planung und Konfiguration, um eine maximale Leistung und Genauigkeit zu gewährleisten. Jedoch existiert für Apache Flink eine aktive Gemeinschaft, welche die Plattform aktualisiert und ständig weiter entwickelt.[48] Zusätzlich bietet Flink dem Benutzer ein reduziertes Maß an Komplexität durch die Integration traditioneller Datenbankkonzepte wie deklarative Abfragesprachen.[48]

Apache Beam mit Flink

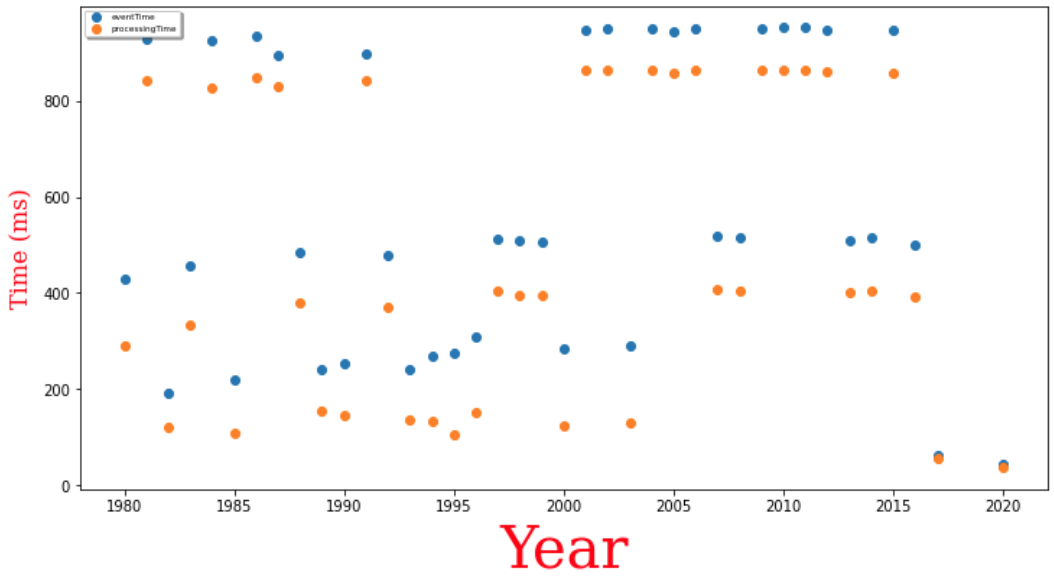

Abbildung 5.8: Apache Beam mit dem Flink Runner

Der Apache Beam Job mit dem Flink Runner weist eine Event-Time-Latenz von bis zu 950ms auf. Die Processing-Time-Latenz liegt bei maximal 850ms. Im Jahr 2017 und 2020, wo nur sehr wenige Werte zu einem Zustand vorhanden sind, liegt die Event-Time-Latenz bei nur ca.20-40ms. Bei Zuständen, welche viele Werte enthalten, wie die Jahre 2008 und

2009, fallen jedoch keine Besonderheiten auf. Die Latenzzeiten für diese Zustände liegen im Durchschnitt.

Apache Beam mit Spark

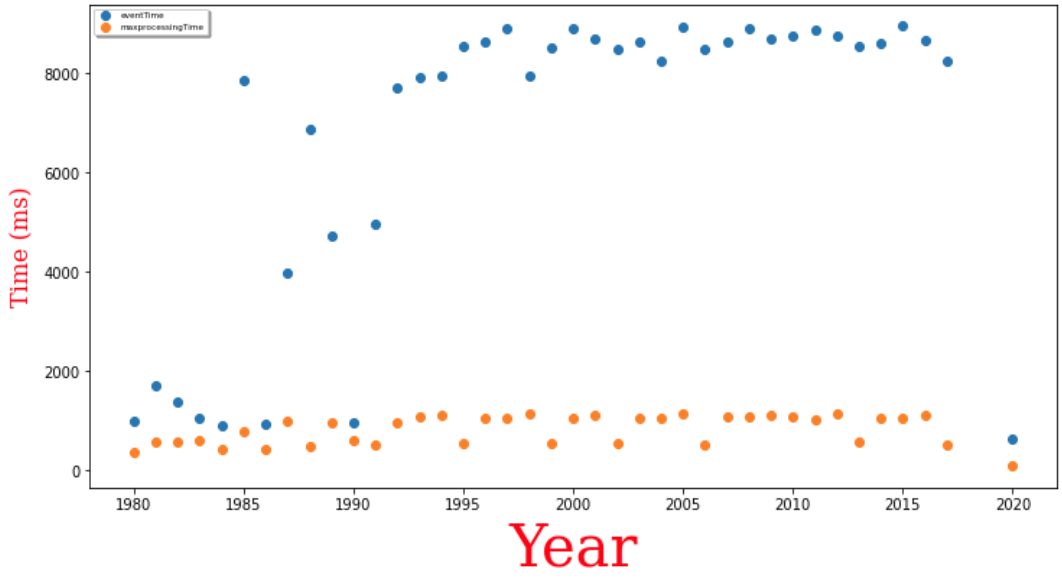

Abbildung 5.9: Apache Beam mit dem Spark Runner

Die Event-Time-Latenz der Apache Beam Pipeline mit dem Spark Runner benötigt bis zu 10 Sekunden. Der Job weist eine Processing-Time-Latenz von 0.3 bis zu 2 Sekunden auf. Hier wird nicht die maximale Verarbeitungszeit wie in den anderen getesteten Systeme zurückgegeben, sondern nur die Verarbeitungszeit des letzten Ereignisses. Siehe: 4.5.9

Wie in der Abbildung: 5.9 zu sehen ist, benötigen die Zustände mit wenigen Ereignissen erheblich kürzer für die Verarbeitung, wodurch auch eine geringere Event-Time Latenz entsteht. Für den Zustand das Jahr 2020 beträgt die Event-Time Latenz z.B. nur 500ms.

Komplexität Apache Beam

Für Apache Beam muss nur eine Pipeline implementiert werden, welche mit verschiedenen Runnern ausgeführt werden kann. Zudem ist es möglich, die Pipelines in Apache Beam

mit verschiedenen Programmiersprachen zu entwickeln. Dies reduziert die Komplexität. Es existieren viele Informationen über Apache Beam und die Community von Apache Beam wächst.

Die Pipeline in Apache Beam weist 62 Programmierzeilen auf. In Flink werden 93 Programmierzeilen für die CombineFn Funktion benötigt. In Spark weist diese nur 83 Programmierzeilen auf, da diese Funktion nur die letzte Verarbeitungszeit der Ereignisse berechnet. Die benutzerdefinierte Klasse weist 120 Programmierzeilen auf und der Serialisierer für diese Klasse 21 Programmierzeilen.

5.2 Auswertung

In diesem Abschnitt werden die Ergebnisse der Event-Streaming-Verarbeitungssysteme ausgewertet und es werden die Forschungsfragen beantwortet.

5.2.1 Latenz

Um die zweite Forschungsfrage:
Wie schnell ist die Verarbeitung und die Ereignislatenz der verschiedenen Ereignisse innerhalb der Event-Streaming-Verarbeitungssysteme?(Siehe: 4.1)
und einen Teil der dritten Forschungsfrage:
Wie wirkt sich die Nutzung von Apache Beam auf die Latenzzeiten aus, steigen die Latenzzeiten durch die Abstraktionsschicht?(Siehe: 4.1)
zu beantworten, wurde der der Durchschnitt der Latenzen für jedes Event-Streaming-Verarbeitungssystem berechnet. Hierfür wurden die Zeiten der einzelnen Zustände summiert und anschließend wurde die Summe durch die Anzahl der Zustände dividiert.

Storm, Kafka-Streams und der Flink Job, welcher in Java implementiert worden ist, weisen die geringsten Latenzzeiten auf. Dieses Streaming Systeme lesen, verarbeiten und schreiben die Daten in nahezu Echtzeit. Flink und Kafka benötigen ungefähr dieselbe Zeit für den gesamten Prozess wie für die Verarbeitung. Daraus lässt sich schließen, dass das Lesen und Schreiben der Daten in diesen Systemen extrem performant ist. Die Serialisierung und Deserialisierung der Ereignisse spielen hier eine große Rolle. KsqlDB benötigt für die Verarbeitung nur bis zu 0.1 Sekunden, das Lesen und Schreiben der Daten dauert ein wenig, sodass die Event-Time-Latenz bei ca. einer Sekunde liegt. Spark hingegen

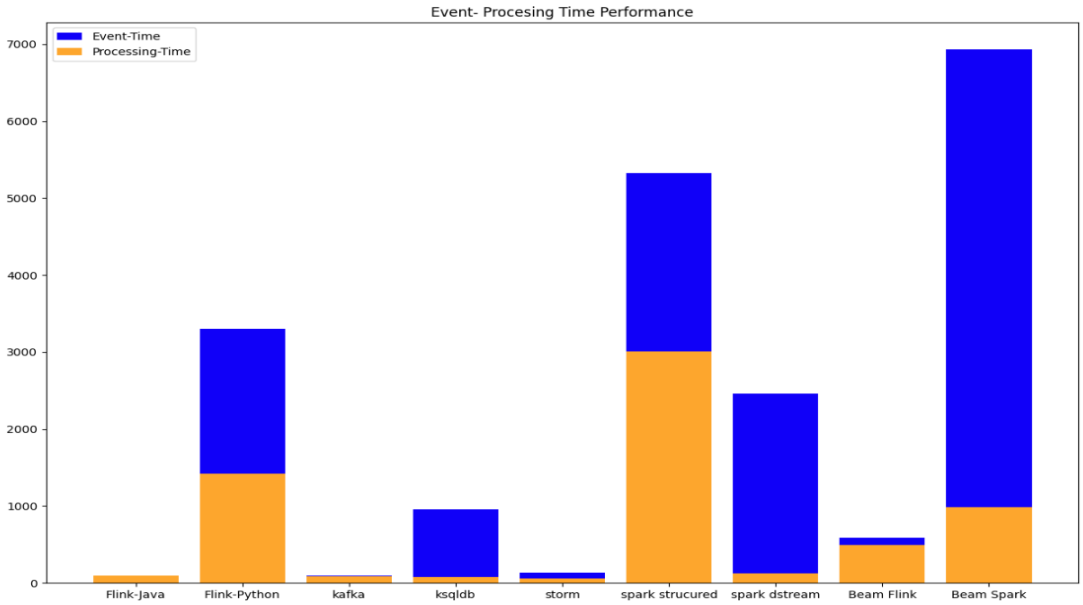

Abbildung 5.10: Event und Processing Time von allen Systemen

benötigt im Schnitt bis zu fünf Sekunden für den gesamten Job. Selbst die Verarbeitung dauert länger als der gesamte Prozess bei Storm, Kafka, Flink und KsqlDB. Dies liegt zum Teil an der verwendete Technik von Spark, da dieses System mit Mikro-Batches arbeitet und Spark Structured Streaming standardmäßig einen Trigger verwendet, welcher alle 500ms eine Berechnung des generierten Mikro-Batches anstößt.

Apache Spark ermöglicht die Implementierung von Stream Verarbeitungen mithilfe von zwei verschiedenen Abstraktionen. Bei der Auswertung hat die Spark Structured Streaming API eine höhere Processing-Time-Latenz als die Spark Streaming API aufgewiesen. Die Event-Time-Latenz ist bei beiden Spark Anwendungen nahezu identisch ist und liegt bei ungefähr 2 Sekunden. Die unterschiedliche Verarbeitungszeit bei Spark liegt an der Implementierung, es gibt einige DataFrame/Dataset-Operationen, die bei Streaming Anwendungen nicht unterstützt werden. Eine davon ist, dass mehrere Streaming-Aggregationen, eine Kette von Aggregationen auf einem DataFrame noch nicht unterstützt werden. Daher war es nicht möglich mit einer SQL-Abfrage, die globalen Verkäufe zu aggregieren, die maximale Verarbeitungszeit zu berechnen und die letzte zugehörige 'Append log Time' vom Kafka-Broker zu erhalten. Um dies mit Spark Structured Streaming umzusetzen, mussten die Daten zunächst gruppiert werden und mit einer MapGroupsWithStateFunction verarbeitet werden. Diese Funktion benötigt laut dem Experiment mehr Zeit, als die updateStateByKey Funktion der Spark Streaming

API. Falls keine Verarbeitungszeit sowie die letzte zugehörige 'Append log Time' benötigt wird, ist es möglich, diesen Job mit einer SQL-Abfrage zu implementieren, welche durch den Catalyst-Optimierer eine deutlich geringere Latenz aufweisen würden. Der Catalyst-Optimierer optimiert strukturelle Abfragen, welche in SQL oder über die DataFrame/Dataset-APIs ausgedrückt werden. Daher ist es wichtig, vorher zu wissen, welchen Zweck die Stream-Verarbeitungs-Anwendungen erfüllen soll, um zu prüfen, ob dies mit der Schnittstelle des Stream-Verarbeitungssystems möglich ist.

Es ist möglich, eine Spark-Streaming-Anwendung so zu konfigurieren, dass diese performanter wird. Dies ist durch Parameter und einem ganzheitlichen Verständnis des gesamten Systems möglich. Es ist z.B. möglich, die Größe der Treiber- und Executorspeicher zu erweitern oder auch den Anwendungscode zu verbessern. Wichtig ist hier die richtige Auswahl der Funktionen und Parameter für den jeweiligen Anwendungsfall. Jedoch handelt es sich bei Spark nicht um eine native Streaming-Verarbeitungsplattform. Stattdessen handelt es sich um eine schnelle Batch-Verarbeitung, welche mit kleinen Batches der eingehenden Daten arbeitet. Dies fällt besonders in Bereichen der Leistung und Zustandsverwaltung auf.

Abstraktionen können sich jedoch auch negativ auf die Leistung des Systems auswirken. Dies ist in 5.10 gut zu erkennen. PyFlink ist um ein Vielfaches langsamer als derselbe Job, der mit Flink in Java implementiert worden ist. Es ist zu erkennen, dass der Overhead an Kommunikation zwischen Python und P4j den Job erheblich verlangsamt. Die Leistung des Jobs hängt jedoch vollständig von der Art der Implementierung ab. Wenn z.B. eine benutzerdefinierte Funktion (UDF) in Python geschrieben wird und diese in Flink ausgeführt wird, wird die Leistung langsamer sein als bei der Verwendung von Scala/Java-basiertem Code. Dies liegt hauptsächlich an der Konvertierung von Python-Objekten und umgekehrt. Die Konvertierung muss durchgeführt werden und kostet Ressourcen. Zusätzlich ist Python eine interpretierte Sprache, daher ist sie langsamer in der Ausführung im Vergleich zu anderen Programmiersprachen, welche direkt kompiliert werden. Python eignet sich nicht für umfangreiche Aufgaben, die viel Speicherplatz benötigen. Bei einer interpretierten Sprache wird der Quellcode nicht direkt von der Zielmaschine übersetzt.[40] Stattdessen liest ein anderes Programm, der sogenannte Interpreter, den Code und führt diesen aus.

In der Abbildung:5.10 ist sehr gut zu erkennen, dass die Anwendungen in Beam viel höhere Latenzzeiten aufweisen. Der Job, welcher in Apache Beam entwickelt worden ist und mit dem Apache Flink Runner ausgeführt worden ist, weist geringer Latenzzeiten

auf als derselbe Job, welcher nur mit PyFlink implementiert worden ist. Daraus lässt sich schließen, dass der Overhead von Apache Beam mit dem Flink Runner geringer ist als die Kommunikation zwischen Python und Java mithilfe von Py4j. Apache Beam mit dem Spark Runner benötigt am längsten, um die Ereignisse zu lesen, zu verarbeiten, zu aggregieren und zu schreiben. Die Abstraktionen kosten Ressourcen, wodurch höhere Latenzzeiten entstehen.

Beim Experiment wurden die standard Windows der Verarbeitungssysteme benutzt, bei Spark Streaming wird z.B. alle 500ms ein Mikro-Batch zur Verarbeitung ausgelöst. Apache Flink, Kafka-Streams und KsqlDB aktualisieren die Ausgabe eines Windows immer dann, wenn das Window ein neues Ereignis erhält. Um einen Vergleich zwischen Apache Beam und den getesteten Event-Streaming-Verarbeitungssystemen herstellen zu können, wurde in Apache Beam ein Trigger definiert, welcher nach jedem empfangenen Ereignis die Aggregation anstößt. In 5.10 ist gut zu erkennen, dass sowohl Flink als auch Spark wesentlich performanter sind als dieselben Anwendungen mit Apache Beam und dem jeweiligen Runner. Dies liegt zum Teil an der Übersetzung durch den Runner und unterscheidet sich je nach dem verwendeten Runner. Jede Funktion in der Apache Beam Pipeline muss in eine kompatible Funktion in dem entsprechenden Stream-Verarbeitungssystem gemappt werden. Bei der Verwendung von z.B. Python in Beam müssen zusätzliche Umgebungen erstellt werden, sodass der Code ausgeführt werden kann. Dies kostet weitere Ressourcen und führt zu höheren Latenzzeiten. Es ist wichtig zu erwähnen, dass Apache Beam den nativen Systemen hinterher hängt. Die Runner müssen die Aktualisierungen der nativen Stream-Verarbeitungssysteme zunächst implementieren, bevor diese in Apache Beam benutzt werden können.

Apache Beam mit Flink ist jedoch für die implementiere Aufgabe schneller als derselbe Job in PyFlink, Spark Streaming, Spark Strucutured Streaming oder KsqlDB.

5.2.2 Komplexität

Damit die erste Forschungsfrage:
Erleichtert eine hohe Abstraktionsschicht die Implementierung der Event-Streaming-Jobs in den Verarbeitungssystemen?(Siehe: 4.1)
und der zweite Teil der dritten Forschungsfrage:
Wie wirkt sich die Nutzung von Apache Beam auf die Komplexität aus, sinkt die Komplexität durch die Abstraktionsschicht?(Siehe: 4.1)

beantwortet werden können, wurden die Programmierzeilen gezählt und der Aufwand, um den Job zu implementieren, bewertet sowie die Größe der 'Community' herangezogen.

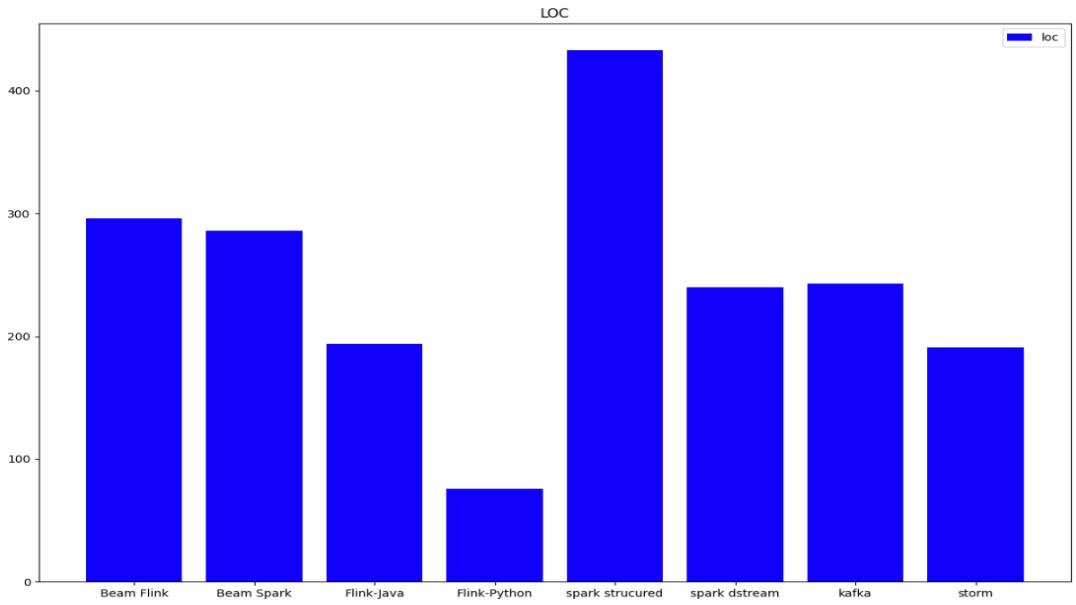

Abbildung 5.11: Alle LOC

Durch die Verwendungen von Abstraktionen in Event-Stream-Verarbeitungssystemen wird die Komplexität des Aufbaus und des Betriebs dieser Systeme verringert. Die Leistung, Skalierbarkeit und Zuverlässigkeit wird durch die Verwendung von Abstraktionen verbessert. Dies geschieht durch die Bereitstellung von einfacheren, übergeordneten Schnittstellen und z.B. der Datenabstraktion, wodurch Event-Streaming-Verarbeitungssysteme es dem Anwender ermöglichen, robuste und skalierbare Anwendungen zu implementieren und gleichzeitig den Aufwand für die Verwaltung, Transformation oder Aggregation der Daten sowie die Verwaltung der Infrastruktur verringern.

Wie in 5.11 zu sehen ist, benötigt der Python Job für Apache Flink am wenigsten Programmierzeilen. Der Streaming-Job, welcher mit Spark Structured Streaming implementiert worden ist und dieselbe Aufgabe erfüllt, benötigt ca. das Vierfache an Programmierzeilen. Die restlichen Anwendungen benötigen ungefähr dieselbe Anzahl an Programmierzeilen, um die Aufgaben des Experimentes zu implementieren. KsqlDB benötigt nur vier SQL-Statements, um die Daten zu erhalten, die Zeitstempel hinzuzufügen und diese zu verarbeiten. Daher wurden in KsqlDB keine Programmierzeilen gezählt.

Durch die Möglickeit andere Sprachen als Java für die Implementation zu verweden, kann die Komplexität verringert werden. Der Einstieg in eine Programmiersprache wie Python ist einfacher als der Einstieg in z.B. Java. [40] KsqlDB ermöglicht die Implementierung mit einer SQL-ähnlichen Syntax. Dadurch entfällt die Notwendigkeit eine komplexe Programmiersprache wie Java zu beherrschen, um eine Streaming-Anwendung zu implementieren. Die Komplexität wird in Apache Beam durch den portablen Runner und der dazugehörigen FN-API verringert, da es hierdurch möglich ist, meherere Programmiersprachen für die Entwicklung von Apache Beam Pipelines zu verwenden.

Der Wechsel in Apache Beam von einem abstrahierten Stream-Verarbeitungssysteme zu einem anderen ist sehr leicht. Bei der Übergabe der Pipeline zu Beam wird der Runner als Parameter übergeben und anschließend ausgeführt. Hierdurch erhöht Beam die Portabilität und Flexibilität. Es ist möglich, die verschiedenen Stream-Verarbeitungssysteme innerhalb von Beam schnell zu vergleichen, um die richtige Kombination aus Umgebung und Leistung für den aktuellen Anwendungsfall zu finden.

Die Implementierung des Streaming-Jobs war in allen Event-Streaming-Systemen aufgrund der großen 'Communities' [32] leicht umzusetzen. Lediglich der Kafka-Streams-Job erfordert die Implementierung in Java oder Scala und bietet keine Schnittstelle zu einer anderen Programmiersprache wie Python.

Durch die Verwendung von Abstraktionen muss sich der Benutzer nicht auf die Details konzentrieren, sondern kann sich vollumfänglich auf die Funktionalität und Logik der Anwendung fokussieren, wodurch die Komplexität reduziert wird.

6 Fazit

Ziel dieser Masterarbeit war es, die verschiedenen Abstraktionen in Event-Streaming-Verarbeitungssystemen zu erforschen und zu evaluieren. Zunächst wurde eine systematische Literaturrecherche zu diversen Event-Streaming-Verarbeitungssystemen und den verwendeten Abstraktionen durchgeführt. Abstraktion ist ein wesentliches Konzept in der Informatik und der Programmierung. Hierdurch ist es möglich, sich auf die relevanten Aspekte des Systems zu konzentrieren, ohne sich mit den Details aufzuhalten.

Durch die Abstraktion von Daten wird eine Trennung zwischen den Daten und den Anwendungen erschaffen, wodurch eine größere Flexibilität, Wartbarkeit und Modularität ermöglicht wird. Zudem kann ein modularer und besser zu wartender Code erstellt und die Leistung verbessert werden.

Abstraktionen können dabei helfen, die Komplexität zu reduzieren. Durch Abstraktionsebenen werden komplexe Systeme in verschiedene Detailstufen unterteilt. Dies trägt dazu bei, dass das Gesamtkonzept vereinfacht und gleichzeitig die wichtigen Details beibehalten werden.

Zusätzlich wird die Komplexität dieser Systeme reduziert, wenn z.B. mehrere Programmiersprachen zur Implementierung zur Auswahl stehen. Durch die Auswahl von mehreren Programmiersprachen wird die Wahrscheinlichkeit erhöht, dass der Entwickler eine dieser Programmiersprachen beherrscht. Bei KsqlDB wird eine SQL-ähnliche Syntax verwendet, die nahezu jeder Entwickler beherrschen sollte. Dies reduziert die Komplexität und den Aufwand für die Implementation enorm.

Die Ereignisse werden in den verschiedenen Event-Streaming-Verarbeitungssystemen unterschiedlich abstrahiert und interpretiert. Die wichtigen Details der Implementierung der Datenabstraktionen wie der Fehlertoleranz, der Unveränderlichkeit oder der Partitionierung, werden vom Anwender versteckt, sodass sich dieser nicht mit den Details befassen muss. Alle getesteten Event-Streaming-Verarbeitungssysteme verstecken diese Details vor dem Anwender. In Apache Spark existieren verschieden Datenabstraktion

wie das Resilient Distributed Dataset oder der DataFrame. Bei der Auswahl der richtigen Datenabstraktion kommt es auf den Anwendungsfall an. Bei unstrukturierten Daten wie Textdaten ist das RDD in Bezug auf die Leistung von Vorteil. Wenn die Daten strukturiert oder halb-strukturiert sind und Abstraktionen auf einer hohen Ebene benutzt werden, bietet Dataframe ein Schema für solche Daten. In Apache Storm ist ein Stream eine Liste von Tupeln und in Kafka-Streams eine geordnete und wiederholbare Folge von unveränderlichen Key-Value-Paaren. Schlüssel werden in Kafka-Streams verwendet, um die Partition innerhalb eines Protokolls zu bestimmen, an die eine Nachricht angehängt wird.[12]

Bevor ein Event-Streaming-Verarbeitungssystem benutzt wird, sollte überprüft werden, ob die Abstraktionen in dem System zu den Anforderungen für die Aufgaben der Stream-Verarbeitung passen. Zudem sollte man die möglichen Funktionen, wie z.B. das State Management geregelt wird, überprüfen, um herauszufinden, ob sich die Aufgabe mit den angebotenen Funktionen umsetzen lässt.

Die Latenzzeiten variieren bei den verschiedenen getesteten Event-Streaming-Verarbeitungssystemen. Kafka-Streams, Apache Flink und Apache Storm lesen, verarbeiten und schreiben die empfangenen Daten mit einer sehr geringen Latenz von 100-250ms. Hier ist besonders beachtlich, dass die Verarbeitung der Daten, die Gruppierung, Aggregation und die Berechnung der maximalen Verarbeitungszeit nur einen sehr kleinen Anteil an der gesamten Latenz aufweist. Lediglich Apache Beam mit dem Flink Runner hat eine ähnliche Distribution. KsqlDB, Apache Flink mit Python(Pyflink), Spark Streaming, Spark Structred Streaming und Apache Beam mit dem Spark Runner weisen eine hohe Verarbeitungszeit auf.

Falls Latenzzeiten akzeptabel sind, welche im Sekundenbereich sind, ist Spark eine gute Wahl. Es ist stabil, hat eine große Gemeinschaft und fast jedes beliebige System kann einfach eingebunden werden.

Die Abstraktionen von Apache Beam haben einen Einfluss auf die Latenzzeiten der Event-Stream-Verarbeitungsjobs. Es ist gut zu erkennen, dass die Pipeline in Apache Beam mit den jeweiligen Runner ein Vielfaches der Latenzzeiten aufweist, die die Pipeline in den nativen Event-Stream-Verarbeitungssystemen benötigt hat. Wenn Apache Beam mit einem gewünschten Runner implementiert werden soll, sollte überprüft werden, ob der Runner die Aufgaben in dem nativen Event-Stream-Verarbeitungssystem ausführen kann. Außerdem ist es wichtig zu überprüfen, ob die Funktionalität so implementiert worden

sind wie die im kohärenten nativen System. Beim Spark Runner wurden z.B. verarbeitete Ereignisse nicht als schon verarbeitet deklariert. Siehe: 4.5.9

Während alle Event-Stream-Verarbeitungssysteme Gemeinsamkeiten in Bezug auf die zugrunde liegenden Konzepte und Arbeitsprinzipien aufweisen, besteht ein wichtiger Unterschied zwischen den einzelnen Systemen, welcher sich direkt auf die Latenzzeiten auswirkt. Die unmittelbare Verarbeitung von Datenelementen bei ihrem Eintreffen minimiert die Latenzzeit und erhöht die Kosten pro Element, während die Pufferung oder die Verarbeitung in Batches oder Mikro-Batches die Latenzzeiten erhöht, jedoch zu einer höheren Effizienz führt.

Zusammenfassend lässt sich dabei als Ergebnis nennen, dass Abstraktionen die Komplexität und den Aufwand für den Entwickler erheblich reduzieren können. Zusätzlich helfen Abstraktionen dabei wie beim Autofahren, sich auf die Funktionalität zu konzentrieren, ohne alle Details zu kennen und zu beachten. Abstraktionen können sich jedoch auch negativ auf die Leistung eines Systems auswirken. Daher sollte man sich mit den verschiedenen zur Auswahl stehenden Abstraktionen beschäftigen und überprüfen, welche für die zu erledigende Aufgabe erforderlich und nützlich sind.

Aussicht

Eine lohnende Aufgabe für die nahe Zukunft ist die Implementation von mehreren Berechnungen, wie zum Beispiel die Zusammenführung von mehreren Datensätzen oder auch die Auswertung von zustandslosen Streaming-Jobs. Eine Evaluation der Datenabstraktion auf Fehlertoleranz und der Skalierbarkeit ist ebenso wünschenswert. Bei der Latenz spielt die Serialisierung und auch die Deserialisierung eine große Rolle, da die Datentypen in Bytes und umgekehrt konvertiert werden müssen. Eine genauere Untersuchung der Serialisierer und Deserialisierer innerhalb der Systeme wäre begrüßenswert. Zusätzlich sollte auch der Durchsatz der Daten innerhalb der Stream-Verarbeitungssysteme getestet werden.

Bei z.B. Apache Spark ist die Konfiguration des Systems ausschlaggebend für viele Metriken. Die Systeme wurden mit ihren standardmäßigen Konfigurationen getestet, daher wäre ein Experiment mit verschiedenen Konfiguration für jedes System sinnvoll.

Windows und Trigger spielen eine große Rolle bei der Berechnung von unendlichen Datensätzen. Eine Analyse der verschiedenen Windows und Trigger innerhalb der Stream-Verarbeitungssysteme ist empfehlenswert.

Außerdem wäre eine weitere Untersuchung, um die Komplexität der Streaming-Jobs in den jeweiligen Systemen zu bestimmen, von Vorteil. Hierfür ist z.B. die Halstaed-Metrik [7] eine geeignete Kennzahl.

Da der Vergleich Grenzen aufweist, weil nur eine Maschine verwendet wird, wäre eine Wiederholung der Experimente mit anderer Hardware sowie mit unterschiedlichen Datensätzen sehr nützlich.

Bei Apache Beam wurde nur der klassiche Runner getestet. Ein Experiment, welches den portablen mit dem klassischen Runner vergleicht, würde einen Mehrwert geben.

Literaturverzeichnis

[1] Akidau, Tyler: Streaming 102: The world beyond batch, URL https://www.oreilly.com/radar/the-world-beyond-batch-streaming-102/, 2016

[2] Akidau, Tyler ; Bradshaw, Robert ; Chambers, Craig ; Chernyak, Slava ; Fernández-Moctezuma, Rafael J. ; Lax, Reuven ; McVeety, Sam ; Mills, Daniel ; Perry, Frances ; Schmidt, Eric ; Whittle, Sam: The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, out-of-Order Data Processing. In: *Proc. VLDB Endow.* 8 (2015), aug, Nr. 12, S. 1792–1803. – URL https://doi.org/10.14778/2824032.2824076. – ISSN 2150-8097

[3] Akidau, Tyler ; Chernyak, Slava ; Lax, Reuven: Streaming Systems, O'Reilly Media, Inc., 2018. – URL https://www.oreilly.com/library/view/streaming-systems/9781491983867/. – ISBN 9781491983874

[4] Alcalá-Fdez, Jesús ; Alonso, Jose M. ; Castiello, Ciro ; Mencar, Corrado ; Soto-Hidalgo, José M.: Py4JFML: A Python wrapper for using the IEEE Std 1855-2016 through JFML. In: *2019 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, 2019, S. 1–6

[5] Allen, Sean T. ; Jankowski, Matthew ; Pathirana, Peter: In: *Storm Applied*, URL https://www.manning.com/books/storm-applied, 2015. – ISBN 9781617291890

[6] Antinyan, Vard ; Staron, Miroslaw ; Sandberg, Anna: Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. In: *Empirical Software Engineering* (2017)

[7] Bailey, C. T. ; Dingee, W. L.: A Software Study Using Halstead Metrics. In: *Proceedings of the 1981 ACM Workshop/Symposium on Measurement and Evaluation*

of Software Quality. New York, NY, USA : Association for Computing Machinery, 1981, S. 189–197. – URL https://doi.org/10.1145/800003.807928. – ISBN 0897910389

[8] Beam, Apache: . – URL https://beam.apache.org/documentation/programming-guide/#triggers

[9] Beam, Apache: Apache Beam Capability Matrix, URL https://beam.apache.org/documentation/runners/capability-matrix/

[10] Beam, Apache: Apache Beam Dokumentation, URL https://beam.apache.org/get-started/beam-overview/

[11] Beam, Apache: Apache Beam Dokumentation Runner, URL https://beam.apache.org/documentation/runners/

[12] Bejeck, Bill: Kafka Streams in action, Manning Publications. – URL https://livebook.manning.com/book/kafka-streams-in-action/about-this-book. – ISBN 978-1617298684

[13] Bordin, Maycon V. ; Griebler, Dalvan ; Mencagli, Gabriele ; Geyer, Cláudio F. R. ; Fernandes, Luiz Gustavo L.: DSPBench: A Suite of Benchmark Applications for Distributed Data Stream Processing Systems. In: *IEEE Access* 8 (2020), S. 222900–222917

[14] Bricks, Data ; Chambers, Bill ; Zaharu, Matei: Apache Spark and Delta Lake Under the Hood, URL https://www.databricks.com/resources/ebook/apache-spark-delta-lake-under-the-hood, 2018

[15] Carbone, Paris ; Katsifodimos, Asterios ; Ewen, Stephan ; Markl, Volker ; Haridi, Seif ; Tzoumas, Kostas: Apache Flink™: Stream and Batch Processing in a Single Engine, URL https://asterios.katsifodimos.com/assets/publications/flink-deb.pdf, 2015

[16] Chandramouli, Badrish ; Goldstein, Jonathan ; Barga, Roger ; Riedewald, Mirek ; Santos, Ivo: Accurate latency estimation in a distributed event processing system, 04 2011, S. 255–266

[17] Chauchot, Etienne: The journey of building a Beam runner based on Spark structured streaming framework, URL https://www.youtube.com/watch?v=oEehQwOEFvg

[18] CHAWLA, MEENU ; BANIWAL, VINITA: Optimization in the catalyst optimizer of Spark SQL. In: *Turkish Journal of Electrical Engineering and Computer Sciences* (2018). – URL https://doi.org/10.3906/elk-1707-6

[19] CHEN, Zeqiang ; CHEN, Nengcheng ; GONG, Jianya: Design and implementation of the real-time GIS data model and Sensor Web service platform for environmental big data management with the Apache Storm. In: *2015 Fourth International Conference on Agro-Geoinformatics (Agro-geoinformatics)*, 2015

[20] CONFLUENT: Confluent Processing-guarantees. . – URL https://docs.confluent.io/platform/current/streams/concepts.html#processing-guarantees

[21] DATA BRICKS, Jules D.: A Tale of Three Apache Spark APIs: RDDs vs DataFrames and Datasets. (2016). – URL https://www.databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

[22] DEAN, Jeffrey ; GHEMAWAT, Sanjay: MapReduce: Simplified Data Processing on Large Clusters. (2004). – URL https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

[23] DEVOPEDIA: Apache Beam. Version 4, URL https://devopedia.org/apache-beam, 2020

[24] FLINK, Apache: Api Concepts Flink, URL https://nightlies.apache.org/flink/flink-docs-release-1.8/dev/api_concepts.html, 2015

[25] FLINK, Apache: Table Concepts Flink, URL https://nightlies.apache.org/flink/flink-docs-release-1.8/dev/table, 2015

[26] FLINK, Apache ; MICHELS, Maximilian ; SFIKAS, Markos: How Beam runs on top of Flink, URL https://flink.apache.org/2020/02/22/apache-beam-how-beam-runs-on-top-of-flink/, 2020

[27] GARCÍA-GIL, D. ; RAMÍREZ-GALLEGO, S. ; GARCÍA, S. et a.: A comparison on scalability for batch big data processing on Apache Spark and Apache Flink, 2016

[28] GOOGLE: Protocol Buffers - Google's data interchange format, URL https://github.com/protocolbuffers/protobuf

[29] HARUNA, ISAH ; TARIQ, ABUGHOFA ; SAZIA, MAHFUZ ; DHARMITHA, AJERLA ; ZULKERNINE, FARHANA ; SHAHZAD, KHAN: A Survey of Distributed Data Stream Processing Frameworks. (2019). – URL https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8864052

[30] HESSE, Guenter ; LORENZ, Martin: Conceptual Survey on Data Stream Processing Systems. In: *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, 2015

[31] HESSE, Guenter ; MATTHIES, Christoph ; GLASS, Kelvin ; HUEGLE, Johannes ; UFLACKER, Matthias: Quantitative Impact Evaluation of an Abstraction Layer for Data Stream Processing Systems. In: *CoRR* abs/1907.08302 (2019). – URL http://arxiv.org/abs/1907.08302

[32] INFORMATIK, FZI F.: Smart-Data-Technologien, URL https://www.digitale-technologien.de/DT/Redaktion/DE/Downloads/Publikation/Smart_Data_Technologien.pdf?__blob=publicationFile&v=6, 2015

[33] IQBAL, Muhammad ; SOOMRO, Tariq: Big Data Analysis: Apache Storm Perspective. In: *International Journal of Computer Trends and Technology* 19 (2015), 01, S. 9–14

[34] JAIN, Ankit: Mastering Apache Storm. . – URL https://learning.oreilly.com/library/view/mastering-apache-storm/9781787125636

[35] KAFKA: Kafka Streams Dokumentation, Kafka, 2022. – URL https://kafka.apache.org/10/javadoc/org/apache/kafka/streams/

[36] KAFKA: Kafka Streams Dokumentation ARCHITECTURE, Kafka, 2022. – URL https://kafka.apache.org/0102/documentation/streams/architecture

[37] KAGGLE, URL https://www.kaggle.com/datasets/gregorut/videogamesales

[38] KARIMOV, Jeyhun ; RABL, Tilmann ; KATSIFODIMOS, Asterios ; SAMAREV, Roman ; HEISKANEN, Henri ; MARKL, Volker: Benchmarking Distributed Stream Data Processing Systems. (2018), 04, S. 1507–1518

[39] Karimov, Jeyhun ; Rabl, Tilmann ; Katsifodimos, Asterios ; Samarev, Roman ; Heiskanen, Henri ; Markl, Volker: Benchmarking Distributed Stream Data Processing Systems. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, S. 1507–1518

[40] Khoirom, Mrs. S. ; Sonia, Moirangthem ; Laikhuram, Borishphia ; Laishram, Jaeson ; Singh, Tekcham D.: Comparative Analysis of Python and Java for Beginners, URL https://www.semanticscholar.org/paper/Comparative-Analysis-of-Python-and-Java-for-Khoirom-Sonia/d787d59179cc56b412fd880fe407b3c1cbca71c5, 2020

[41] Kramer, Jeff: Is Abstraction the Key to Computing? In: *Commun. ACM* 50 (2007), apr, Nr. 4, S. 36–42. – URL https://doi.org/10.1145/1232743.1232745. – ISSN 0001-0782

[42] KsqlDB: Data definition, KsqlDB, 2022. – URL https://docs.ksqldb.io/en/latest/operate-and-deploy/capacity-planning/

[43] KsqlDB: Plan Capacity, KsqlDB, 2022. – URL https://docs.ksqldb.io/en/latest/reference/sql/data-definition/

[44] Liskov, Barbara: Keynote Address - Data Abstraction and Hierarchy. In: *SIGPLAN Not.* 23 (1987), jan, Nr. 5, S. 17–34. – URL https://doi.org/10.1145/62139.62141. – ISSN 0362-1340

[45] Lopez, Martin A. ; Lobato, Antonio Gonzalez P. ; Duarte, Otto Carlos M. B.: A Performance Comparison of Open-Source Stream Processing Platforms. In: *2016 IEEE Global Communications Conference (GLOBECOM)*, 2016, S. 1–6

[46] Qadeer, Abdul ; Heidemann, John: Performance Optimizations and Operator Semantics for Streaming Data Flow Programs. (2020). – URL https://edoc.hu-berlin.de/bitstream/handle/18452/22285/dissertation_sax_matthias%20j..pdf?sequence=7&isAllowed=y

[47] Qadeer, Abdul ; Heidemann, John: Efficient Processing of Streaming Data using Multiple Abstractions. In: *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021, S. 157–167

[48] Rabl, Tilmann ; Traub, Jonas ; Katsifodimos, Asterios ; Markl, Volker: Apache Flink in current research. In: *it - Information Technology* 58 (2016), 01

[49] Salloum, S. ; Dautov, R. ; Chen, X. et a.: Big data analytics on Apache Spark. (2016)

[50] Seymour, Mitch: In: *Mastering Kafka Streams and ksqlDB*, O'Reilly Media, Inc., 2021. – ISBN 9781492062493

[51] Spark, Apache: Apache Spark Dokumentation. . – URL https://spark.apache.org/docs/latest/

[52] Spæren, Teodor: Programming model for Apache Beam, URL https://cloud.google.com/dataflow/docs/concepts/beam-programming-model

[53] Spæren, Teodor: Performance Analysis and Improvements for Apache Beam, URL https://www.duo.uio.no/bitstream/handle/10852/87850/1/teodor_spaeren_master.pdf, 2021

[54] Storm, Apache: Apache Storm Dokumentation. . – URL https://storm.apache.org/releases/current/javadocs/org/apache/storm/

[55] V, Srinivas J. ; P, Srikanth ; Krishnamachari, Thumati ; Sri, Hari N.: A Review Study of Apache Spark in Big Data Processing. (2016). – URL https://www.academia.edu/25701979/_IJCST_V4I3P16_V_Srinivas_Jonnalagadda_P_Srikanth_Krishnamachari_Thumati_Sri_Hari_Nallamala

[56] Wojtczyk, Martin ; Knoll, Alois: A Cross Platform Development Workflow for C/C++ Applications, URL https://mediatum.ub.tum.de/doc/1289361/906486.pdf, 2008

[57] Zaharia, Matei ; Chowdhury, Mosharaf ; Das, Tathagata ; Dave, Ankur ; Ma, Justin ; McCauley, Murphy ; Franklin, Michael J. ; Shenker, Scott ; Stoica, Ion: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. USA : USENIX Association, 2012 (NSDI'12), S. 2

[58] Zaharia, Matei ; Xin, Reynold S. ; Wendell, Patrick ; Das, Tathagata ; Armbrust, Michael ; Dave, Ankur ; Meng, Xiangrui ; Rosen, Josh ; Venkataraman, Shivaram ; Franklin, Michael J. ; Ghodsi, Ali ; Gonzalez, Joseph ; Shenker, Scott ; Stoica, Ion: Apache Spark: A Unified Engine for Big Data Processing. (2016). – URL https://doi.org/10.1145/2934664. – ISSN 0001-0782

A Anhang

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original